

Nucleus C++ NET

Reference Manual



0001132-001 Rev 100

Copyright (c) 2002
Accelerated Technology, Inc.
720 Oak Circle Dr. E.
Mobile, AL 36609
(251) 661-5770



Related Documentation

Nucleus PLUS Reference Manual, by Accelerated Technology, describes the operation and usage of the Nucleus PLUS kernel.

Nucleus PLUS Internals, by Accelerated Technology, describes, in considerable detail, the implementation of the Nucleus PLUS kernel.

Nucleus NET Reference Manual, by Accelerated Technology, describes the operation and usage of the Nucleus NET component.

Nucleus Config Reference Manual, by Accelerated Technology, describes the operation and usage of the Nucleus Config component.

Nucleus C++ BASE Reference Manual, by Accelerated Technology, describes the operation and usage of the Nucleus C++ BASE component.

Nucleus C++ PLUS Reference Manual, by Accelerated Technology, describes the operation and usage of the Nucleus C++ PLUS component.

Style and Symbol Conventions

Program listings, program examples, filenames, menu items/buttons and interactive displays are each shown in a special font.

Program listings and program examples - Courier New

Filenames - COURIER NEW, ALL CAPS

Interactive Command Lines - **Courier New, Bold**

Menu Items/Buttons – *Times New Roman Italic*

Trademarks

MS-DOS is a trademark of Microsoft Corporation

UNIX is a trademark of X/Open

IBM PC is a trademark of International Business Machines, Inc.

Additional Assistance

For additional assistance, please contact us at the following:

Accelerated Technology
720 Oak Circle Drive, East
Mobile, AL 36609
800-468-6853
251-661-5770
251-661-5788 (fax)

support@atinucleus.com

<http://www.acceleratedtechnology.com>

Copyright (©) 2002, All Rights Reserved.

Document Part Number : 0001132-001 Rev 100

Last Revised: January 31, 2001





Contents

Chapter 1-Nucleus C++ NET User's	1
About this Manual.....	2
What is in this User's guide?.....	2
What is Nucleus C++ NET?	2
Nucleus C++ NET Class Hierarchy	3
Nucleus C++ NET Source Files.....	4
Portable Source Files.....	4
Target Dependent Source Files	5
Application Dependent Source Files	5
Optimizations for Embedded Systems.....	5
Network Initialization and Startup.....	5
Initializing the NppNET Component	6
Registering Network Devices.....	6
Static Objects.....	6
Nucleus C++ NET Socket Classes.....	7
Two Part Socket Object Creation.....	7
Using TCP Sockets (class NuTcpSocket).....	7
Client Connecting to a Server 1-2-3	8
Construct the NuTcpSocket Object.....	8
Create the NuTcpSocket Object.....	8
Connect to the Server.....	8
Sending Data.....	9
Receiving Data.....	10
Close the Connection.....	11
Server Accepting a Connection from a Client 1-2-3	11
Construct the NuTcpSocket Object.....	11
Create the NuTcpSocket Object.....	11
Bind to the Desired IP Address and Port	12
Listen for Clients	12
Accept Client Connection.....	13
Sending and Receiving Data.....	13
Close the Connection.....	13



Using UDP sockets (class NuUdpSocket)	13
Receiving Data from a Client (Server role) 1-2-3	14
Construct the NuUdpSocket Object	14
Create the NuUdpSocket Object	14
Bind the Local Address to the Socket	14
Wait to Receive Data from a Client	15
Optionally Send Data to the Client	15
Close the Socket.....	15
Sending Data to a Server (Client Role) 1-2-3	16
Construct the NuUdpSocket Object	16
Create the NuUdpSocket Object	16
Send Data to a Server.....	16
Optionally Receive Data from the Server	17
Close the Socket.....	17
Using Raw IP Sockets (class NuRawSocket)	17
Sending Data to a Server (Client Role) 1-2-3	17
Construct the NuRawSocket Object.....	17
Create the NuRawSocket Object.....	18
Send Data to a Server.....	18
Optionally Receive Data from the Server	18
Close the Socket.....	18
Receiving Data from a Client (Server Role) 1-2-3	19
Construct the NuRawSocket Object.....	19
Create the NuRawSocket Object.....	19
Wait to Receive Data from a Client	19
Optionally Send Data to the Client	19
Close the Socket.....	19
Nucleus C++ NET Devices	20
“C” or C++?	20
Class NuNetDevice	21
Registering your Device with the Network Stack	21
Deriving your own Network Device 1-2-3	21
Write a LISR to Receive Data.....	22
Write an Init() Routine	22
Write a Start() Routine	23
Write an Ioctl() Routine	23
Write an Input() Routine	23
Write an Output() Routine	24
Class WrapperDevice	24
Nucleus C++ NET SIMPLE Demonstration Application	25
Why aren't the Nucleus C++ NET Device Classes Demonstrated?.....	25
Why isn't the Nucleus C++ PLUS Component Used?.....	26
Files and File Locations	26
Hardware Reference Design Setup	27
Downloading and Executing the Demo	28
Application Details.....	28



Common Application Settings, Data, and Procedures	28
Startup and Initialization	30
Application Tasks	35
Chapter 2-Nucleus C++ NET class Reference	47
class IPAddress	48
Public Member Functions	48
Protected Member Functions	49
IPAddress::~~IPAddress	49
IPAddress::Get	50
IPAddress::IPAddress	51
IPAddress::IPAddress	52
IPAddress::IPAddress	53
IPAddress::IPAddress	54
IPAddress::IsClassA	55
IPAddress::IsClassB	56
IPAddress::IsClassC	57
IPAddress::IsClassD	58
IPAddress::MaskClassA	59
IPAddress::MaskClassB	60
IPAddress::MaskClassC	61
IPAddress::MaskClassD	62
IPAddress::operator &=	63
IPAddress::operator []	64
IPAddress::operator []	65
IPAddress::operator =	66
IPAddress::operator =	67
IPAddress::operator =	68
IPAddress::operator const struct id_struct*()	69
IPAddress::operator struct id_struct*	70
IPAddress::Set	71
IPAddress::Set	72
class NppNET : public NppComponent	73
Public Member Functions	73
Protected Member Functions	73
NppNET::~~NppNET	74
NppNET::Initialize	75
NppNET::InitializeClasses	76
NppNET::InitializeNetwork	77
NppNET::NppNET	78
NppNET::RegisterDevice	79
NppNET::UnRegisterDevice	80



class NuNetDevice	81
Public Member Functions	81
Protected Member Functions	82
NuNetDevice::~~NuNetDevice	83
NuNetDevice::GetDeviceEntryStruct	84
NuNetDevice::GetDeviceStruct	85
NuNetDevice::GetDeviceStruct	86
NuNetDevice::GetDriverOptions	87
NuNetDevice::GetFlags	88
NuNetDevice::GetGateway	89
NuNetDevice::GetIP	90
NuNetDevice::GetName	91
NuNetDevice::GetSubnet	92
NuNetDevice::Init	93
NuNetDevice::Input	95
NuNetDevice::Ioctl	97
NuNetDevice::NuNetDevice	99
NuNetDevice::Output	100
NuNetDevice::SetDriverOptions	102
NuNetDevice::SetFlags	103
NuNetDevice::SetGateway	104
NuNetDevice::SetInitRoutine	105
NuNetDevice::SetInputRoutine	107
NuNetDevice::SetIP	109
NuNetDevice::SetName	110
NuNetDevice::SetOutputRoutine	111
NuNetDevice::SetSubnet	113
NuNetDevice::Start	114
class NuRawSocket : public NuSocket	115
Public Member Functions	115
NuRawSocket::~~NuRawSocket	116
NuRawSocket::Create	117
NuRawSocket::NuRawSocket	119
NuRawSocket::RecvFromRaw	120
NuRawSocket::RecvFromRaw	122
NuRawSocket::SendToRaw	123
NuRawSocket::SendToRaw	125
NuRawSocket::SetAppLayerDoesIPHeader	127
class NuSocket	129
Public Member Functions	129
Protected Member Functions	130
Static Public Class Member Functions	130
NuSocket::~~NuSocket	131
NuSocket::Abort	132
NuSocket::Bind	133



NuSocket::Close	134
NuSocket::Create	135
NuSocket::CreateBase	136
NuSocket::CreateFromSocketDescriptor	137
NuSocket::EnableBroadcasting	139
NuSocket::Fcntl	140
NuSocket::FD_Check	141
NuSocket::FD_Init	143
NuSocket::FD_Reset	144
NuSocket::FD_Set	146
NuSocket::GetMulticastTTL	147
NuSocket::GetPeerAddress	148
NuSocket::GetPeerAddress	149
NuSocket::GetSocketDescriptor	150
NuSocket::GetSocketFor	151
NuSocket::Getsockopt	153
NuSocket::Initialize	155
NuSocket::IsDataAvailable	156
NuSocket::IsValid	157
NuSocket::NuSocket	158
NuSocket::operator=	159
NuSocket::Ping	160
NuSocket::SetBlocking	162
NuSocket::SetSocketDesc	163
NuSocket::Setsockopt	164
NuSocket::WaitForData	166
class NuTcpSocket : public NuSocket	168
Public Member Functions	168
NuTcpSocket::~~NuTcpSocket	169
NuTcpSocket::Accept	170
NuTcpSocket::Accept	172
NuTcpSocket::Connect	174
NuTcpSocket::Connect	176
NuTcpSocket::Create	178
NuTcpSocket::EnableTcpDelay	180
NuTcpSocket::IsConnected	182
NuTcpSocket::Listen	183
NuTcpSocket::NuTcpSocket	185
NuTcpSocket::Recv	186
NuTcpSocket::Send	187
class NuUdpSocket : public NuSocket	188
Public Member Functions	188
NuUdpSocket::~~NuUdpSocket	189
NuUdpSocket::Create	190
NuUdpSocket::NuUdpSocket	192



NuUdpSocket::RecvFrom	193
NuUdpSocket::RecvFrom	194
NuUdpSocket::SendTo	197
NuUdpSocket::SendTo	199
class SocketAddress	201
Public Member Functions	201
SocketAddress::~~SocketAddress	202
SocketAddress::GetFamily	203
SocketAddress::GetIP	204
SocketAddress::GetIP	204
SocketAddress::GetIPBits	205
SocketAddress::GetName	206
SocketAddress::GetPort	207
SocketAddress::operator =	208
SocketAddress::operator =	209
SocketAddress::operator const struct addr_struct*	210
SocketAddress::operator struct addr_struct*	211
SocketAddress::Set	212
SocketAddress::SetFamily	213
SocketAddress::SetIP	214
SocketAddress::SetIP	215
SocketAddress::SetIP	216
SocketAddress::SetIP	217
SocketAddress::SetName	218
SocketAddress::SetPort	219
SocketAddress::SocketAddress	220
SocketAddress::SocketAddress	221
SocketAddress::SocketAddress	222
SocketAddress::SocketAddress	223



1

Nucleus C++ NET User's Guide

About this Manual

What is C++ NET?

Nucleus C++ NET Class Hierarchy

Nucleus C++ NET Source Files

Example Code

Optimizations for Embedded
Systems

Network Initialization and Startup

Nucleus C++ NET socket classes

Using TCP sockets

Using UDP sockets

Using RAW sockets

Nucleus C++ NET Device Drivers



About this Manual

This manual is intended to give readers a high level overview of Nucleus C++ NET so they are better prepared to take advantage of its specific design goals and features.

What is in this User's guide?

This guide contains explanations of the various source files, what demos are included, application startup, and the various classes included in Nucleus C++ NET.

What is Nucleus C++ NET?

Nucleus C++ NET consists of a management layer for a C++ class interface into Nucleus NET, a TCP/IP protocol stack specifically designed to work in embedded systems in conjunction with the Nucleus PLUS real-time operating system. The interface is modeled on the defacto standard for IP networking, the Berkeley sockets interface.

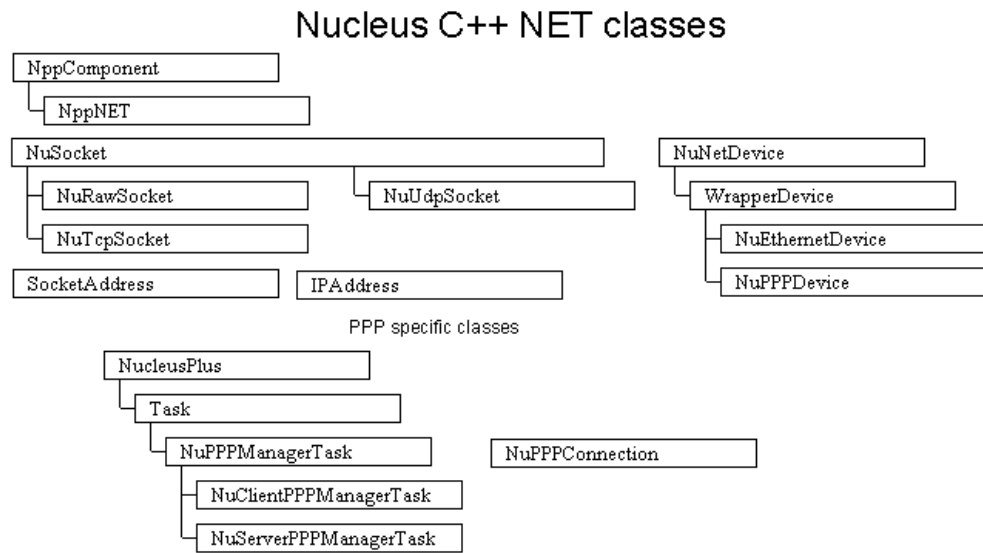
Generally speaking, a socket is just an endpoint of network communication. Nucleus C++ NET provides an intuitive approach to using sockets by encapsulating repeated steps for the creation and use of sockets, as well as packaging all of the functionality that sockets have in easy to use classes.

A general discussion of networking and network protocol stacks is beyond the scope of this manual, but if you are interested in better understanding networking concepts and implementation, we whole-heartedly recommend that you read Andrew Tanenbaums' excellent book "Computer Networking" (ISBN 0-13-349945-6) published by Prentice-Hall which contains a thorough treatment of layered network architecture.



Nucleus C++ NET Class Hierarchy

The following diagram shows the overall Nucleus C++ NET class hierarchy.

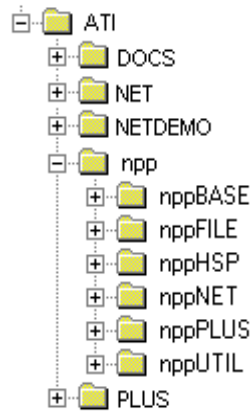


Nucleus C++ NET Class Hierarchy



Nucleus C++ NET Source Files

Nucleus C++ NET source files are delivered in the \NPPNET sub-directory under the Nucleus C++ \NPP directory.



Nucleus C++ NET directory is nppNET

Portable Source Files

In the NPP\NPPNET sub-directory you will find the following portable files. All Nucleus C++ NET targets share these files and they are maintained on a product wide basis.

File	Description
NPPNET.H NPPNET.CPP NPPNET.INL	Source files for class NppNET and Nucleus C++ NET global routines.
NPPSOCK.H NPPSOCK.CPP NPPSOCK.INL	Source file for classes: IPAddress, SocketAddress, NuSocket, NuTcpSocket, NuUdpSocket, and NuRawSocket.
NPPNETDV.H NPPNETDV.CPP NPPNETDV.INL	Source files for classes: NuNetDevice, WrapperDevice.
NPPENDV.H NPPENDV.CPP NPPENDV.INL	Source files for class: NuEthernetDevice.



Target Dependent Source Files

There are not any target dependent files in Nucleus C++ NET. If for some reason a particular version of Nucleus C++ requires special software, the convention is that the files will be named `NPPNETS.H`, `NPPNETS.INL`, and `NPPNETS.CPP` and located in the `NPP\NPPNET` directory. Please refer to the target specific notes for your version of Nucleus C++ NET to check if there are any target specific files.

Application Dependent Source Files

In the `NPP\NPPBASE` sub-directory you will find an application dependent file, `NPPAPP.H`. Each Nucleus C++ application will have a different version of this file. It is used to specify application specific parameters for all Nucleus C++ components used in that particular application.

File	Description
<code>NPPAPP.H</code>	This file contains all application tuning settings for Nucleus C++. By setting various preprocessor switches, you can include or exclude various features from the build to minimize the footprint of your application.

Optimizations for Embedded Systems

There are not any optimization options specific to Nucleus C++ NET. However, there are many options that you can configure within the Nucleus NET TCP/IP stack itself. Please refer to the Nucleus NET documentation for a discussion of these settings and the impact that they have on your application.

Network Initialization and Startup

In the Nucleus C++ BASE user's guide you will find a discussion of the Nucleus C++ component architecture. Please read this first for an understanding of the Nucleus C++ component architecture and how startup works in a Nucleus C++ system. Here we will limit our discussion to Nucleus C++ NET startup and initialization.



Initializing the NppNET Component

The NppNET component is responsible for the initialization of the underlying Nucleus NET TCP/IP stack as well as managing any network devices that are added by the application. It can be created on either the system startup thread or the multitasking startup thread, but the `Initialize` member must be called on a multitasking thread since it does the actual work of initializing the networking stack (a requirement of Nucleus NET).

Below is a typical example of initialization code. Note the call to a routine that creates and registers the network device is called after the Nucleus C++ NET component has been initialized.

```
void
NppCreateMultitasking( void* first_available_memory )
{
    ... other Nucleus C++ components created and initialized here.

    // Nucleus C++ NET component.
    NppNET* nppNET = new NppNET();
    nppNET->Initialize();

    // Create and register the devices for the network component.
    NppInitializeEthernetDevices( nppNET );

    // Nucleus C++ STATIC component.
    #if (NU_STATIC_OBJECT_SUPPORT)
        NppSTATIC* nppSTATIC = new NppSTATIC();
        nppSTATIC->Initialize();
    #endif

    ... setup your application here.
}
```

Registering Network Devices

In the preceding code example, you will notice the call to a function called `NppInitializeEthernetDevices`. There is nothing particularly special about this function other than its role. Before you can send or receive data, a networking device must be created and registered with the Nucleus C++ NET component. Please see the section on network devices for more information.

Static Objects

You will notice `NppSTATIC` is the last Nucleus C++ support component initialized prior to any application code. This is done since `NppSTATIC` is responsible for iterating the list of static objects within the system and calls the constructors of these objects one-by-one. Since the `NppNET` component has already been initialized, full networking facilities are available to all static objects within the system, including services that are accessed within constructors.



Nucleus C++ NET Socket Classes

Nucleus C++ provides classes for three different types of sockets that all share a common base class.

Socket class	Description
NuSocket	An abstract base class that is never used directly, but only through the derived classes. It provides all functionality and data that is common to all sockets.
NuTcpSocket	Class for doing connection oriented TCP communications.
NuUdpSocket	Class for doing connectionless UDP communications.
NuRawSocket	Class for doing connectionless communications at the IP level. (bypassing the transport mechanisms that UDP and TCP provide).

Two Part Socket Object Creation

Creating a new socket is a two part process: construction then creation. The constructors for the classes simply initialize data members. They do not interact with the network stack. After the object is constructed, the virtual member `Create` is called which does the actual work of allocating a socket descriptor for the object within the network stack. Why the separation?

If we performed the allocation within the constructor and you had a `NuTcpSocket` object as a member of a class, you would have to assure the network was properly initialized before your object was constructed. Also, as soon as your socket connection was aborted or closed, your object would be useless because you would not be able to reinvoke the constructor to allocate a new socket descriptor.

Using TCP Sockets (class `NuTcpSocket`)

TCP communications are connection based, and as such require that applications decide upon roles (who is the *client* and who is the *server*). Clients initiate connections with servers that are assumed to be waiting for clients to connect.



Client Connecting to a Server 1-2-3

The steps and code required for each step are outlined below.

Construct the NuTcpSocket Object

NuTcpSocket objects can be constructed anywhere you like: local on the stack, global at file scope, or dynamically via a call to operator new.

```
// Local or file scope instance.
NuTcpSocket theSocket;

// Allocate dynamically from the heap.
NuTcpSocket* pSocket = new NuTcpSocket;
```

Create the NuTcpSocket Object

Invoking Create on the socket object does the actual work of allocating a socket descriptor from the Nucleus NET stack for the object.

```
// Allocate a socket descriptor from the TCP/IP stack.
theSocket.Create();    // On an object or reference.
pSocket->Create();     // On a pointer to an object.
```

Connect to the Server

To connect to a server, you must call the Connect method. This member takes as an argument either a pointer or a reference to a SocketAddress object that contains socket address information for the server. It returns a zero or positive value if successful, and a negative Nucleus NET status code otherwise.

```
SocketAddress serverAddress;
NuTcpSocket theSocket;

serverAddress.SetIP( 209, 67, 27, 72 );
serverAddress.SetPort( 80 );

if( theSocket.Create() )
{
    STATUS status = theSocket.Connect( serverAddress );
    if( status >= 0 )
    {
        // Yes! we are connected
    }
    else
    {
        // rats! we didn't make the connection. We can find out why
        // by examining status.
    }
}
```



Sending Data

After a connection has been established, data can be sent and received at will. Which side (client or server) is the sender or receiver is entirely an application decision. The following example shows a typical send data operation.

```
UINT8  data_buffer[100];
UINT16 size_to_send;
INT32  bytes_sent;

bytes_sent = theSocket.Send( data_buffer, size_to_send );

// Check results.
if( bytes_sent != size_to_send )
{
    // Could not send the data, check if it is due to a lost connection.
    if( bytes_sent == NU_NOT_CONNECTED )
    {
        // No longer connected.
        connected = FALSE;
    }
    else
    {
        error("Cannot send.");
    }
}
```



Receiving Data

There are two modes under which we can receive data: blocking and non-blocking. When in blocking mode, the receive data operation will suspend the calling thread until data has been received or the connection has been terminated. In non-blocking mode, the calling thread is not suspended. The mode can be changed using the member `SetBlocking`. The following example shows a typical receive data operation.

```
UINT8 data_buffer[100];
INT32 bytes_received;

theSocket.SetBlocking( FALSE ); // Turn off blocking on a read.

bytes_received = theSocket.Recv( data_buffer, 100 );

if( bytes_received > 0 )
{
    // We received good data.
}
else if( bytes_received == 0 )
{
    // We didn't receive any data, but we are still connected.
}
else
{
    // The value is negative meaning that a Nucleus NET status code was
    // returned indicating a problem.
    if( bytes_received == NU_NOT_CONNECTED )
    {
        // The connection was dropped.
    }
}
```

You will notice there is not an optional timeout value on the receive call. You either unconditionally suspend or you don't. What do you do if you want to timeout if data hasn't arrived after a certain interval of time? Simply call `WaitForData` which takes a number of timer ticks to wait for data and then returns a status code. If data is available, you can then call `Recv` to retrieve it.



Close the Connection

Closing the connection is important because it returns resources to the network stack. A connection can be closed one of two ways in a TCP connection. Calling the member `Close` elegantly attempts to close the connection, returning once the other side of the connection has also closed.

The other way is to call `Abort`, which immediately and unconditionally shuts down the connection and sends a reset to the other side without waiting for it to close.

```
if( theSocket.Close() == NU_SUCCESS )
{
    // Swell, it worked.
}
```

Server Accepting a Connection from a Client 1-2-3

The steps and code required for each step are outlined below.

Construct the NuTcpSocket Object

`NuTcpSocket` objects can be constructed anywhere you like: local on the stack, global at file scope, or dynamically via a call to operator `new`.

```
// Local or file scope instance.
NuTcpSocket theSocket;

// Allocate dynamically from the heap.
NuTcpSocket* pSocket = new NuTcpSocket;
```

Create the NuTcpSocket Object

Invoking `Create` on the socket object does the actual work of allocating a socket descriptor from the Nucleus NET stack for the object.

```
// Allocate a socket descriptor from the TCP/IP stack.
theSocket.Create();    // On an object or reference.
pSocket->Create();     // On a pointer to an object.
```



Bind to the Desired IP Address and Port

Servers must tell the stack which IP address and port number that they want to wait for connections on. To do this, call `Bind`.

```
NuTcpSocket  theSocket;
SocketAddress thisAddress;

// Allocate a socket descriptor from the TCP/IP stack.
if( theSocket.Create() )
{
    // Set the SocketAddress object up with the address
    // information that we want to bind to.
    thisAddress.SetIP( 200,100,50, 1 );
    thisAddress.SetPort( 8002 );

    if( theSocket.Bind( thisAddress ) >= 0 )
    {
        // We were able to bind the address.
```

Listen for Clients

After `Bind` has been called, the next step is calling `Listen` which sets up the number of simultaneous connection requests that can be queued for the server.

```
if( theSocket.Bind( thisAddress ) >= 0 )
{
    // We were able to bind the address.

    // Queue a maximum of 16 connection requests for the server.
    theSocket.Listen( 16 );
```



Accept Client Connection

After the previous steps have all been done, we are completely ready to accept connections from clients via a call to `Accept`. Similar to `Recv`, `Accept` can be called in either blocking or non-blocking mode. The mode can be set via a call to the `SetBlocking` method. The default behavior is to block waiting for clients to connect. As an alternative, you can wait with a timeout using `WaitForData`.

```
// Queue a maximum of 16 connection requests for the server.
theSocket.Listen( 16 );

// We want to suspend waiting for clients to connect. This is the
// default, but we do it here just for clarities sake.
theSocket.SetBlocking( TRUE );

// Construct a new "worker" socket to handle this connection.
// Note that you do not have to do the "Create" step in this case
// because the network stack does it for you.
SocketAddress clientAddress;
NuTcpSocket* pWorkerSocket = new NuTcpSocket;

if( theSocket.Accept( pWorkerSocket, &clientAddress ) >= 0 )
{
    // Great! pWorkerSocket is connected to a client.
```

Sending and Receiving Data

After the connection has been established, you can send and receive data at will. The semantics of sending and receiving data are exactly the same between client and server. Please see the example in the previous section.

Close the Connection

Closing the connection is also the same between client and server roles. Please see the example in the previous section.

Using UDP sockets (class NuUdpSocket)

UDP is used for fast *connectionless* communications. This means there isn't the *formal* concept of establishing a connection between a client and a server like there is using TCP. However, this does not mean there isn't the concept of a client and server.

With UDP, a *server* binds to a specific IP address and port, and waits for data on this port. A *client* sends data to this IP address and port, expecting the server to be waiting for the data. When the server receives the data, it also receives the sending clients IP address and port number. The server can then use the address information to send data to the client, assuming the client is waiting for the data.



Therefore, instead of the concept of a connection there is an understanding between communicating nodes as to which of them is the server which of them is the client. These roles can then flip back and forth much like passing notes back in fifth grade.

Please note that constructing a UDP socket, creating it, binding it to an IP address and port, and finally closing it is identical to how it is done for TCP sockets. The difference is how data is sent and received and the missing client step for establishing the connection.

Receiving Data from a Client (Server role) 1-2-3

The steps and code required for each step are outlined below.

Construct the NuUdpSocket Object

NuUdpSocket objects can be constructed anywhere you like: local on the stack, global at file scope, or dynamically via a call to operator new.

```
// local or file scope instance.
NuUdpSocket    theSocket;

// allocate dynamically from the heap
NuUdpSocket* pSocket = new NuUdpSocket;
```

Create the NuUdpSocket Object

Invoking Create on the socket object does the actual work of allocating a socket descriptor from the Nucleus NET stack for the object.

```
// Allocate a socket descriptor from the TCP/IP stack.
theSocket.Create();    // On an object or reference.
pSocket->Create();     // On a pointer to an object.
```

Bind the Local Address to the Socket

Servers must tell the stack which IP address and port number that they want to wait for data on. To do this, call Bind.

```
NuUdpSocket    theSocket;
SocketAddress  thisAddress;

// Allocate a socket descriptor from the TCP/IP stack.
if( theSocket.Create() )
{
    // Set the SocketAddress object up with the address
    // information that we want to bind to.
    thisAddress.SetIP( 64, 27, 126, 2 );
    thisAddress.SetPort( 7 );

    if( theSocket.Bind( thisAddress ) == NU_SUCCESS )
    {
        // We were able to bind the address.
    }
}
```



Wait to Receive Data from a Client

Now that the IP address and port we want has been bound to the socket, we need to wait for data on it. Waiting for and receiving data is done using the `RecvFrom` member. This can be a blocking or non-blocking operation. If you need a timeout, use `WaitForData`.

After the call to `RecvFrom`, the socket address object `clientAddress` encapsulates the IP address and port of the sending client. The integer `actualBytes` either contains the number of bytes received or a negative Nucleus NET status code if there was some kind of problem.

```
if( theSocket.Bind( thisAddress ) == NU_SUCCESS )
{
    // We were able to bind the address, now lets wait
    // for data from the client.
    SocketAddress clientAddress;
    char buffer[2048];
    INT32 actualBytes;

    // Suspending on a read is the default behavior, but we set
    // it here just for clarities sake.
    theSocket.SetBlocking( TRUE );

    // Suspend waiting for data.
    actualBytes = theSocket.RecvFrom( buffer, 2048, clientAddress );
```

Optionally Send Data to the Client

As mentioned above, after the call to `RecvFrom` the `clientAddress` object contains the IP address and port of the sending client. At this point a server will typically parse the data received, do something application specific with it, and then reply to the client with data that is also application specific. Sending the data is accomplished using `SendTo`.

```
// Suspend waiting for data.
actualBytes = theSocket.RecvFrom( buffer, 2048, clientAddress );

// here we assume the existence of a routine that examines the data
// from the client, and formats a response in the same buffer,
// returning the number of bytes in the response.
actualBytes = Parse_Data_Make_Response( buffer );

theSocket.SendTo( buffer, (UINT16)actualBytes, clientAddress );
```

Close the Socket

With UDP, calling the `Close` member does not terminate a connection since there isn't one. Instead, it returns network resources that were allocated for this socket by the stack.

```
theSocket.Close();
```



Sending Data to a Server (Client Role) 1-2-3

The steps and code required for each step are outlined below.

Construct the NuUdpSocket Object

NuUdpSocket objects can be constructed anywhere you like: local on the stack, global at file scope, or dynamically via a call to operator new.

```
// Local or file scope instance.
NuUdpSocket theSocket;

// Allocate dynamically from the heap.
NuUdpSocket* pSocket = new NuUdpSocket;
```

Create the NuUdpSocket Object

Invoking Create on the socket object does the actual work of allocating a socket descriptor from the Nucleus NET stack for the object.

```
// Allocate a socket descriptor from the TCP/IP stack.
theSocket.Create();    // On an object or reference.
pSocket->Create();     // On a pointer to an object.
```

Send Data to a Server

We are assuming when send data to a UDP server that the server exists and is currently waiting for data on the IP address and port we are sending to. If the server is not ready to receive data the send call will fail.

When sending to a server it is not necessary to explicitly bind to an IP address and port. The stack will automatically use the IP address of whatever device in the system has a route to the server and it will find an available port. If you want to use a specific device and port for your outbound data, use Bind as outlined in the server role above.

```
Char          buffer[1600];
NuUdpSocket   theSocket;
SocketAddress serverAddress;
INT32         actualBytes;

theSocket.Create();

serverAddress.SetIP( 192, 200, 100, 3 );
serverAddress.SetPort( 7 );

// format a message to send to the server
messageSize = Format_A_Message( buffer );

// Send the message to the server.
ActualBytes = theSocket.SendTo( buffer, messageSize, serverAddress );
```



Optionally Receive Data from the Server

After you send data to a server, there is a valid IP address and port assigned to the socket. Therefore, we can immediately wait for the server's response using `RecvFrom`.

```
// Send the message to the server.
ActualBytes = theSocket.SendTo( buffer, messageSize, serverAddress );

// Block this thread waiting for the servers response.
actualBytes = theSocket.RecvFrom( buffer, 1600, serverAddress );

// Here the actualBytes contains the number of bytes that were actually
// received, and serverAddress contains the IP and port of the servers
// response (which is important, because the server could have redirected
// us to another IP address and port which handled the request)
```

Close the Socket

Closing the socket is the same as closing all other sockets.

```
theSocket.Close();
```

Using Raw IP Sockets (class `NuRawSocket`)

Raw sockets are a means of using Nucleus NET directly at the network layer, bypassing the transport layers provided by UDP and TCP.

Sending Data to a Server (Client Role) 1-2-3

The steps and code required for each step are outlined below.

Construct the `NuRawSocket` Object

`NuRawSocket` objects can be constructed anywhere you like: local on the stack, global at file scope, or dynamically via a call to operator `new`. Unlike `NuTcpSocket` and `NuUdpSocket` which do not take arguments to their constructors, `NuRawSocket` takes a protocol type. Currently supported values are:

Object	Description
<code>IPPROTO_HELLO</code>	HELLO routing protocol
<code>IPPROTO_OSPF</code>	OSPF routing protocol
<code>IPPROTO_RAW</code>	Raw IP protocol
0	Raw IP wildcard protocol

```
// Create and instance using the raw IP protocol.
NuRawSocket theSocket( IP_PROTO_RAW );
```



Create the NuRawSocket Object

Invoking Create on the socket object does the actual work of allocating a socket descriptor from the Nucleus NET stack for the object.

```
// Allocate a socket descriptor from the TCP/IP stack.  
theSocket.Create();
```

Send Data to a Server

Raw IP is a bit trickier when sending data. It can be used in a manner similar to UDP, but it also provides the ability to send without an IP address. As with UDP, it is assumed that a server is already waiting for us to send data.

```
Char          buffer[1600];  
NuRawSocket   rawSocket( IPRAW_PROT ); // Instantiate for raw protocol.  
SocketAddress serverAddress;  
INT32         actualBytes;  
  
rawSocket.Create();  
  
// Fill out the IP address to send to.  
serverAddress.SetIP( 192, 200, 100, 3 );  
serverAddress.SetPort( 0 ); // Port must be zero.  
  
// Format a message to send to the server.  
messageSize = Format_A_Raw_Message( buffer );  
  
// Send the message to the server.  
actualBytes = rawSocket.SendToRaw( buffer, messageSize, serverAddress );
```

Optionally Receive Data from the Server

Receiving data is accomplished using RecvFromRaw.

```
NuRawSocket   rawSocket( IPRROTO_RAW ); // Instantiate for raw protocol.  
SocketAddress peerAddress;  
INT32         bytesRecieved;  
UINT8         buffer[100];  
  
rawSocket.Create();  
  
// Format a message to send to the server.  
messageSize = Format_A_Raw_Message( buffer );  
  
// Suspend waiting for a message.  
bytesRecieved = rawSocket.RecvFromRaw( buffer, 100, peerAddress );  
  
// At this point, peerAddress contains information on the IP address that  
// the data was sent from.
```

Close the Socket

Closing the socket is the same as closing all other sockets.

```
rawSocket.Close();
```



Receiving Data from a Client (Server Role) 1-2-3

The steps and code required for each step are outlined below.

Construct the NuRawSocket Object

NuRawSocket objects can be constructed anywhere you like: local on the stack, global at file scope, or dynamically via a call to operator new.

```
// Create and instance using the raw IP protocol.
NuRawSocket theSocket( IP_PROTO_RAW );
```

Create the NuRawSocket Object

Invoking Create on the socket object does the actual work of allocating a socket descriptor from the Nucleus NET stack for the object.

```
// Allocate a socket descriptor from the TCP/IP stack.
theSocket.Create();
```

Wait to Receive Data from a Client

Receiving data is accomplished using RecvFromRaw.

```
NuRawSocket    rawSocket( IPRROTO_RAW ); // Instantiate for raw protocol.
SocketAddress  peerAddress;
INT32          bytesRecieved;
UINT8          buffer[100];

rawSocket.Create();

// Format a message to send to the server.
messageSize = Format_A_Raw_Message( buffer );

// Suspend waiting for a message.
bytesRecieved = rawSocket.RecvFromRaw( buffer, 100, peerAddress );

// At this point, peerAddress contains information on the IP address that
// the data was sent from and we can.
```

Optionally Send Data to the Client

Sending data is accomplished using SendToRaw.

```
// Do something with the data that you have received.
INT32 bytesToSend = DoSomething( buffer );
rawSocket.SendToRaw( buffer, bytesToSend, peerAddress );
```

Close the Socket

Closing the socket is the same as closing all other sockets.

```
rawSocket.Close();
```



Nucleus C++ NET Devices

Nucleus NET uses a “C” device driver model that has evolved from the Unix roots of the networking stack. In basic terms, it consists of “C” routines called by Nucleus NET to initialize the device, write complete IP packets into the physical hardware, and remove incoming data to be processed by the upper layers of Nucleus NET. It is assumed the initialization routines for devices create and initialize interrupts.

For device driver developers using Nucleus C++ NET, you have a choice of whether to use the “C” driver model or go in a more object-oriented direction and develop a derived C++ device class.

For More Information

Please see the Nucleus NET documentation for more information about writing your own networking device drivers.

“C” or C++?

With Nucleus C++ NET, you have two options for interaction with network devices:

- 1) Use the Nucleus NET “C” device driver model.
- 2) Use the Nucleus C++ NET device object model.

If you buy a “C” network device driver for your target hardware from Accelerated Technology, Inc., you will get a proven, tested driver written in “C”. For many applications, having a proven driver ready to go makes the most sense, even at the expense of not having a completely object-oriented application.

If on the other hand, a complete object paradigm is important, then it makes sense to spend the time to derive a class from `NuNetDevice` and gain the flexibility and extendibility that comes with well crafted C++ source code.

In case you want to encapsulate an existing Nucleus NET “C” driver, a wrapper class derived from `NuNetDevice` is created and instances of this class are registered with the Nucleus C++ NET component. This allows you to gain the benefit of objects without completely writing a networking device driver. Please see the section on class `WrapperDevice` below.



Class NuNetDevice

Nucleus C++ NET manages devices through a C++ class interface. Specialized networking devices are represented by classes derived from NuNetDevice. Instances of a NuNetDevice derived device class are registered with the NppNET component which inserts them into the network stack.

Derived device classes overload member functions of the base class to do the actual work of sending and receiving packets to and from the hardware.

Registering your Device with the Network Stack

After the NppNET component is created, devices must be registered with it before the network is used. The example below is for a PPP specific device class.

```
// Create the Nucleus C++ NET component and initialize it.
NppNET* pNetworkComponent = new NppNET;
pNetworkComponent->Initialize();

// Create a PPP networking device.
pPPPDevice = new NuPPPDevice
(
    "PPP_Link",    // device name
    COM2,          // comPort
    19200,         // baudRate
    PARITY_NONE,   // parity
    DATA_BITS_8,  // dataBits
    STOP_BITS_1    // stopBits
);

// If you need to configure any additional attributes of the device object,
// that is done here before the call to RegisterDevice.
if( pNetworkComponent->RegisterDevice( pPPPDevice ) )
{
    // Excellent! We are ready to do some networking!
}
else
{
    // Tragedy, the registration failed.
}
```

Deriving your own Network Device 1-2-3

Below you will find the steps that are required to write your own Nucleus C++ NET networking device. There are a few pure virtual routines in the base class NuNetDevice that you must define and there are a number of optional virtual routines that you may want to overload.



Write a LISR to Receive Data

With the exception of a few polled devices, most implementations have a hardware interrupt that goes off when data is ready to be received from the hardware. In some cases, such as with Ethernet, the chipset you use will give you an entire packet each time. In other cases, such as PPP over a modem, you have to packet and check the framing in your interrupt handler one byte at a time.

Any interrupts that you use should be configured within the `Init` routine.

You will end up writing an interrupt that takes your data, packets it up, and then passes it to Nucleus NET to be handled. Here is the sequence you must follow:

- 1) Read incoming data from your hardware.
- 2) Strip off any hardware specific framing so you are left with raw IP packets.
- 3) Allocate a Nucleus NET buffer chain big enough to hold all your new data.
- 4) Copy the data into the Nucleus NET buffer chain.
- 5) Notify Nucleus NET that there is new data in the buffers that needs to be handled.

Write an `Init()` Routine

```
Virtual STATUS Init() = 0;
```

`Init` is a pure virtual routine that must be overloaded in a derived `NuNetDevice` class. Nucleus NET calls this routine when a device is registered. Its role is to prepare the device for use by Nucleus NET.

Prior to the call to `Init`, Nucleus NET takes the `NU_DEVICE` structure (which is a member of `NuNetDevice`) and copies the relevant fields into a newly allocated instance of `DV_DEVICE_ENTRY`. This is the structure Nucleus NET uses internally to refer to devices.

Within `Init`, you should initialize any required function pointers and data fields in the `DV_DEVICE_ENTRY` structure, preferably using the base class `NuNetDevice` members. This makes it easier and safer since the members shield you from possible device structure changes in the underlying Nucleus NET “C” device driver model.

In addition to device *instance* initialization, any one time device *type* initialization should be performed. This includes registering any necessary LISR's or HISR's with the Nucleus PLUS kernel. For instance, if you were designing a product that had 10 different physical devices and all were the same type, `Init` could use the concept of lazy initialization to spot the first time it was being called. Of course, this is an optional technique and you can utilize whatever mechanism makes sense for your class initialization.

When `Init` exits, the device should be ready to transmit and receive data. Return `NU_SUCCESS` if all goes well or a negative Nucleus NET status code if there is a problem.



Write a Start() Routine

```
Virtual STATUS Start( NET_BUFFER* pBuffer ) = 0;
```

Start is a pure virtual member that you must define for your derived device class. Nucleus NET calls the Start function to transmit a packet. It is passed a complete packet by the upper layer software and its responsibility is to place the packet onto the physical medium. The parameter is a pointer to the buffer chain containing the packet.

In this routine you should do whatever is necessary to transmit a packet. That may mean placing the data to be transmitted in queue by means of a transmit interrupt you registered in Init, or it can mean writing the data into a buffered hardware chipset.

NU_SUCCESS should be returned upon success. A negative Nucleus NET status code should be returned if an error or some kind of failure occurs.

Write an Ioctl() Routine

```
Virtual STATUS Ioctl( INT option, DV_REQ* pDeviceRequest ) = 0;
```

Ioctl is a pure virtual routine that you must define. IOCTL requests are the means by which UNIX applications traditionally control device drivers. They are used by Nucleus NET for a couple of things, most notably, the addition and deletion of multicast addresses to the list of multicast addresses that are to be received by the driver. The defines for these requests are DEV_ADDMULTI and DEV_DELMULTI. If you don't want to add multicast support, simply return a negative Nucleus NET status code.

Write an Input() Routine

```
virtual STATUS Input();
```

Input is a virtual routine that is called by default for you by Nucleus NET to demultiplex packets from memory buffers, decode any framing that your device may have on the packets, and pass them to the upper layers of Nucleus NET by calling IP_Interpret.

It is assumed by Nucleus NET that one copy of Input is used for each different MAC layer. That means, for example, that all Ethernet devices should share the same input routine. In fact, for Ethernet devices, this routine is provided by Nucleus NET and is called NET_Ether_Input.

If you want to call a different input handler rather than the default implementation that calls this virtual Input member, you will probably create another static member and call SetInputRoutine within Init, passing in a pointer to the static member.



Write an Output() Routine

```
Virtual STATUS Output
(
    NET_BUFFER* pBuffer, SCK_SOCKADDR_IP* pDest, RTAB_ROUTE* pRoute
);
```

Like Input, it is assumed that there is one copy of an output routine that is common to all devices that share a given MAC layer. The default implementation is to call this virtual Output routine which you can overload if desired.

Typically, Output dequeues outgoing IP packets for the Nucleus NET buffers, performs hardware specific framing, and then calls Start for the device corresponding to a given route for actual physical transmission of the packet. For Ethernet devices, this has also been provided by Nucleus NET. The routine is named NET_Ether_Send.

Like we discussed in the input section above, you may want to provide your own implementation and call SetOutputRoutine within Init with a pointer to a static member.

Class WrapperDevice

WrapperDevice is derived from NuNetDevice and is provided to create an easy way to use existing Nucleus NET “C” networking drivers with Nucleus C++ NET. As mentioned above, you can also use existing Nucleus NET “C” drivers using the “C” API for installation and management. WrapperDevice is here to allow the application to take a complete object-based approach to the design, including the networking device aspect of the design.

To use it, you must set all relevant fields within the underlying device structure, most importantly the initialization routine for the device driver. Below is an example that uses the existing Nucleus NET NE2000 Ethernet “C” device driver.

```
// The device must exist at file scope or be newed out of the heap.
pDevice = new WrapperDevice;

pDevice->SetInitRoutine( NE2000_init );
pDevice->SetName( "NE2000" );
pDevice->SetIP( IPAddress( 192, 200, 100, 1 ) );
pDevice->SetEthernetIrq( 10 );
pDevice->SetEthernetIOAddress( 0x300 );

// If you need to configure any additional attributes of the device object,
// that is done here before the call to RegisterDevice.
if( nppNET->RegisterDevice( pDevice ) )
{
    // Excellent! We are ready to do some networking!
}
```



Nucleus C++ NET SIMPLE Demonstration Application

The Nucleus C++ NET demonstration application, `SIMPLE`, demonstrates the basic Nucleus C++ NET classes `NppNET`, `IPAddress`, `SocketAddress`, `NuTcpSocket`, and `NuUdpSocket`. Both client and server roles of network sockets are demonstrated.

The demo is designed to communicate with the host-side Winsock application `NETDEMO\NETDEMO.EXE`, which runs on any Windows 95/98 or Windows NT machine that is connected to the network.

Overview

`SIMPLE` parallels the standard `NETDEMO` application that ships with Nucleus NET in order to take advantage of the features offered by this demo. The goal is to reuse the infrastructure of the “C” `NETDEMO` application, including the network and driver initialization module, the Nucleus Config application, and the host side Nucleus `NETDEMO` Winsock application.

The `SIMPLE` Nucleus C++ NET demonstration application is written using the Nucleus C++ NET classes as opposed to the “C” Nucleus NET API. Otherwise, the application is identical to its “C” counterpart. Please note however that the TFTP portion of the `NETDEMO` application is not available because it is not applicable to Nucleus C++ NET.

For More Information

Details about the `NETDEMO` application and how to use it are included in the Nucleus NET reference manual. Details about the Nucleus Config application can be found in its reference manual. Please refer to these documents for more information.

Why aren't the Nucleus C++ NET Device Classes Demonstrated?

The advantage of reusing the infrastructure of the “C” `NETDEMO` application is that it allows the Nucleus C++ NET `SIMPLE` demo to run out-of-the-box without any modifications on future releases of Nucleus NET for new hardware reference designs. In addition, as soon as new Nucleus NET device drivers are developed and ported to the Nucleus NET `NETDEMO` platform, they are immediately available for use as the driver for the `SIMPLE` application.

The disadvantage of this approach is that the Nucleus C++ NET device classes are not demonstrated since network initialization is performed using the Nucleus NET “C” API.



Why isn't the Nucleus C++ PLUS Component Used?

By examining the source files for the Nucleus C++ NET SIMPLE demo, you will notice the *tasking*, *inter-process communication (queue)*, and *event* mechanisms use the Nucleus PLUS “C” API as opposed to using Nucleus C++ PLUS classes.

While we agree that it would be much nicer for Nucleus C++ users to see a completely object-oriented application that uses the Nucleus C++ PLUS classes `Task`, `Pipe`, and `Event`, not all users of Nucleus C++ NET have this optional Nucleus C++ component. Therefore, the threading and other aspects of the design are implemented in “C” and are not object-oriented. This demonstration application focuses instead on the object-oriented use of the Nucleus C++ NET socket classes.

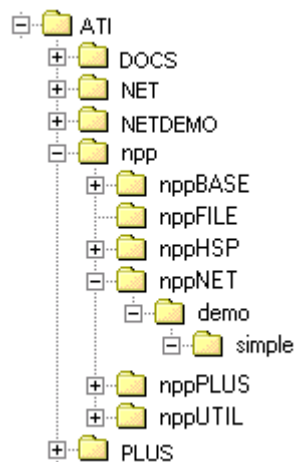
Files and File Locations

The Nucleus C++ NET SIMPLE demonstration application includes two parts, the portable application part and the target specific implementation part. The portable application relies on the target specific NETDEMO hardware support package library.

There are several versions of the NETDEMO hardware support package library, each corresponding to specific reference design needs. However, all of them create a common library, named `NPPHSP_DEMO_NETD.LIB`. The application is linked with this library.

Application Files

The Nucleus C++ NET SIMPLE demonstration application source files are located in the `NPP\NPPNET\DEMO\SIMPLE` subdirectory. In this directory, you will also find the build files required to build the application using your specific tools. The build files will be named `NPPNET_DEMO_SIMPLE.*` where ‘*’ will be specific to the tools used to create the application. For example, the application that is built using Nucleus EDE has build files named `NPPNET_DEMO_SIMPLE.EDE` and `NPPNET_DEMO_SIMPLE.DSP`.



The Simple Nucleus C++ NET Demo Directories



Source File	Description
NETD.H	This is a common header file that contains shared settings that the NETD.CPP module and the various target specific hardware support packages require.
NETD.CPP	The main implementation file for the demo, including the initialization points for the application (NppCreate and NppCreateMultitasking).

Hardware Support Packages Files

The various Nucleus C++ NET SIMPLE demo hardware support package source files are located in the NPP\NPPHSP subdirectory. In this directory, you will also find the build files required to build the libraries using your specific tools. The build files will be named NPPHSP_DEMO_NETD_*. * where ‘*. *’ will be a combination specific to the target reference design and the tools used to create the library. For example, the hardware support package that allocates networking buffers from the C++ freestore and is built using Nucleus EDE has build files named NPPHSP_DEMO_NETD_FREESTORE.EDE and NPPHSP_DEMO_NETD_FREESTORE.DSP.

For specific notes on your hardware support package, please see the target specific notes for Nucleus C++ NET that is shipped with your target version.

Source File	Description
NPPND???.CPP	The hardware support package source files for the SIMPLE demo application will be named using this filename convention. For example, the implementation file that allocates networking buffers from the C++ freestore is named NPPNDFS.CPP.

Hardware Reference Design Setup

The hardware reference design setup required to execute this application is identical to the hardware reference design setup for executing the “C” NETDEMO application for the particular underlying Nucleus NET reference design. Please refer to the Nucleus NET target specific notes for information on setting up the hardware.

Since this SIMPLE demo does not use the standard Nucleus C++ demonstration I/O subsystem, it is possible that the hardware setup to execute the Nucleus C++ BASE demos will be different. For example, if you are using the PPP driver, a modem might be installed on the port that the Nucleus C++ BASE demo uses for output. Of course, please use the hardware setup required to execute the “C” NETDEMO application instead of the one used for the Nucleus C++ BASE demo.



Downloading and Executing the Demo

Downloading and executing the `SIMPLE` application is the same as downloading and executing the Nucleus NET demonstration application. Please refer to the Nucleus NET target specific notes for more information.

Application Details

This section steps through the `SIMPLE` demo, discussing the details as they relate to Nucleus C++ NET in order to highlight the different approach to network programming using object-oriented sockets.

The demo is made up of several application tasks that demonstrate how to use both TCP sockets and UDP sockets in both client and server roles. Tasks includes a TCP server, a TCP client, a UDP server, a UDP client, a host-side communications task, and a loopback TCP client.

Common Application Settings, Data, and Procedures

The shared settings, data, and procedures that are common to the application are found in both the header file and at the top of the portable implementation file.

Constants

The various `#define` macros define buffer sizes, tasking attributes like stack sizes and priorities, command values that are sent by the host, and command values that are sent from the embedded side.

```
#define UDP_SERVER_MAX_RX_SIZE      1472
#define TCP_SERVER_MAX_RX_SIZE      1460
#define TCP_LOOPBACK_CLIENT_BUFFER_SIZE 32
#define COMM_DATA_LEN               50
#define TASK_STACK_SIZE              3000
#define NORMAL_TASK_PRIORITY         (TM_PRIORITY)
#define COMM_TASK_PRIORITY           (TM_PRIORITY)
#define TCP_CLI_START                1
#define UDP_CLI_START                2
#define TCP_CLI_STOP                 3
#define UDP_CLI_STOP                 4
#define ERRMSG                       "0"
#define INFO                         "1"
```

Inter-process Communication Queue Type

Since the TCP server task will pass sockets to a worker task through an inter-process communication queue, we use a `typedef` for the pointer to a TCP socket that is sent. Only the pointer is communicated.

```
// Type that is sent in inter-process communication queue.
typedef NuTcpSocket* NuTcpSocketPointer;
```



Global Data

The global data includes a control block for the event that is used to indicate that network initialization is complete. Other control blocks are defined for the various tasks in the system and the inter-process communication queue.

Global flags and variables are shared between the host-side communication task and the other tasks in the system. These flags and variables are used to control and synchronize the behavior of the embedded networking tasks. The host-side communication task communicates with the host-side Winsock application, which specifies the values of the shared data and thus indirectly controls the behavior of the application.

```
// Application data.
NU_EVENT_GROUP    Init_Complete;
NU_TASK           TCP_Server_Task_CB;
NU_TASK           TCP_Echo0_Task_CB;
NU_TASK           TCP_Echo1_Task_CB;
NU_TASK           TCP_Client_Task_CB;
NU_TASK           UDP_Server_Task_CB;
NU_TASK           UDP_Client_Task_CB;
NU_TASK           Comm_Task_CB;
NU_QUEUE          socketQueue;
INT               TCP_Stat = 0, UDP_Stat = 0;
UINT32            TCP_Loop = 0, UDP_Loop = 0;
UINT16            TCP_Byte = 0, UDP_Byte = 0;
```

The Host-Side Winsock Application's Socket Address

There is one C++ object included in this shared data. It is a `SocketAddress` object that is shared between the host-side communication task and other tasks in the system and is used to hold the socket address of the host-side Winsock application.

```
SocketAddress      host_address;
```

Synchronizing a Thread with Network Initialization

The routine, `Synch_With_Initialization_Task`, is used to synchronize network initialization within the application. It suspends the calling thread until all NETDEMO initialization tasks are complete.



The initialization event is used to indicate that the DEMOI.C module has completed the initialization of the network stack and all network devices. When the event is set, full networking facilities are available to the rest of the system.

```
VOID Synch_With_Initialization_Task(UNSIGNED argc, VOID *argv)
{
    /* No compilation warnings allowed. */
    UNUSED_PARAMETER(argc);
    UNUSED_PARAMETER(argv);

    // Wait until DEMOI.C completes the initialization.
    UNSIGNED ret_events;
    if
    (
        NU_Retrieve_Events
        (
            &Init_Complete, NU_TRUE, NU_AND, &ret_events, NU_SUSPEND
        )
        !=
        NU_SUCCESS
    )
    {
        error("Coordination event group.");
    }
}
```

Loopback Device

For hardware reference designs that utilize the loopback network device, the following data and functions are used.

```
// Loopback data and function.
#if (INCLUDE_LOOPBACK_DEVICE == NU_TRUE)
    NU_TASK TCP_Loopback_Client_Task_CB;
    VOID TCP_Loopback_Client_Task(UNSIGNED, VOID *);
#endif
```

Startup and Initialization

Since Nucleus NET must be initialized on a *task* thread as opposed to the *system startup* thread, the SIMPLE demo uses NppCreateMultitasking as the application initialization point and NppCreate is empty.

```
void
NppCreate( void* /* first_available_memory */ )
{
    // Perform all initialization on the multitasking startup thread.
    #if !(NU_APPLICATION_INITIALIZE_MULTITASKING)

        // This demo requires the multitasking startup.
        #error Must have NU_APPLICATION_INITIALIZE_MULTITASKING.

    #endif
}
```



The `NppCreateMultitasking` initialization routine creates and initializes the various Nucleus C++ components the SIMPLE demo uses: `NppBASE`, `NppNET`, and `NppSTATIC`. In addition, it creates the Nucleus PLUS event group service the demonstration system uses to synchronize network initialization with the rest of the application. After the target and system specific initialization is complete, the rest of the demonstration application is created and initialized.

```
void
NppCreateMultitasking( void* first_available_memory )
{
    /***** Nucleus C++ components *****/

    // Nucleus C++ BASE component.
    NppBASE* nppBASE = new NppBASE( first_available_memory );
    nppBASE->Initialize();

    /* Create event group for task synchronization. */
    STATUS status = NU_Create_Event_Group( &Init_Complete, "InitDone" );
    if( status != NU_SUCCESS )
    {
        error("Cannot Event Group.");
    }

    // Nucleus C++ NET component.
    NppNET* nppNET = new NppNET();
    nppNET->Initialize();

    // Nucleus C++ STATIC component.
    #if (NU_STATIC_OBJECT_SUPPORT)
        NppSTATIC* nppSTATIC = new NppSTATIC();
        nppSTATIC->Initialize();
    #endif

    /***** Demonstration application *****/
    ... the application initialization code is described below.
}
```

Timing and Synchronization of Network Startup and Initialization

The Nucleus PLUS event group service is created prior to `nppNET`, which is the Nucleus C++ NET component. The event is used to suspend the initialization thread within the `NppNET::Initialize()` member until the `DEMOI.C` module completes network initialization. The Nucleus NET stack and any devices are completely initialized and ready to use after `Initialize` finishes. How is this done?

The component's `Initialize` member calls the routine `NppInitializeNetwork` which is located in the target specific `NETDEMO` hardware support package. Most implementations of `NppInitializeNetwork` create and start the `DEMOI.C` located network initialization task. They then call `Synch_With_Initialization_Task` which does not return until the stack and devices are completely initialized.



You will notice `NppSTATIC` is the last Nucleus C++ support component initialized prior to any application code. This is done since `NppSTATIC` is responsible for iterating the list of static objects within the system and calls the constructors of these objects one-by-one. Since the `NppNET` component has already been initialized, full networking facilities are available to all static objects within the system, including services that are accessed within constructors.

NppInitializeNetwork

Each target specific version of `DEMOI.C` requires external support. Nucleus C++ NET isolates this into the target specific `NETDEMO` hardware support package library. A typical implementation is shown below. This version allocates the networking buffers from the C++ freestore, creates and starts the initialization task, and suspends until initialization is finished.

```
// Global data DEMOI.C needs.
VOID* next_available_memory;
NU_TASK Demo_Init_Task_CB;

// External demo initialization task.
extern "C"
{
    VOID DEMOI_Init_Task( UNSIGNED argc, VOID* argv );
}

STATUS
NppInitializeNetwork()
{
    // Allocate the "next available memory" location for use by DEMOI.C.
    next_available_memory = new char[NU_NET_BUFFER_POOL_SIZE];

    // Create the demo initialization thread. Specify it to start now.
    NU_Create_Task
    (
        &Demo_Init_Task_CB, "dem_init", DEMOI_Init_Task, 0, NU_NULL,
        new char[TASK_STACK_SIZE], TASK_STACK_SIZE,
        NORMAL_TASK_PRIORITY, 0, NU_PREEMPT, NU_START
    );

    // Synchronize with the demo initialization thread. This routine will
    // not return until the DEMOI.C module completes all network
    // initialization and upon its return, full networking facilities
    // are available to the rest of the application.
    Synch_With_Initialization_Task( (UNSIGNED)0, NULL );
}
```



DEMOI.C Network Initialization

The details of target specific initialization within `DEMOI.C` is beyond the scope of this manual. In general however, the `DEMOI_Init_Task` thread found within `DEMOI.C` follows the following pseudo-code pattern:

- Create a memory pool for use by Nucleus NET for networking buffers.
- Call the Nucleus NET initialization routine, `NU_Init_Net`, with a pointer to this memory pool for network stack initialization.
- Initialize a network device driver structure with target specific attributes and call the Nucleus NET device initialization routine, `NU_Init_Devices`, with a pointer to the filled in device structure.
- Perform target and device specific steps required for proper network login. For example, if the networking device is a PPP device the initialization steps might include setting up user-supplied PPP login information, dialing out through a modem to an Internet Service Provider (ISP), and negotiating with the server for proper network login.
- Perform any routing required for the application.
- Set the initialization complete event.

Specifying Your Custom Network Attributes Used by DEMOI.C

The target specific attributes found throughout `DEMOI.C` are generated by the Nucleus NET setup program or by the Nucleus Config utility application that is shipped with your specific version of Nucleus NET. Please refer to the Nucleus NET and Nucleus Config reference manuals for more information.

After installation `DEMOI.C` will be updated with the correct settings for a network driver you choose (or a driver you choose during subsequent sessions of the Nucleus Config application). This file contains all of the target and network specific sections of the demo including IP addresses, Nucleus NET initialization, and network device initialization, and associated parameters like ISP phone numbers and login information.

The IP addresses will need to be set such that they are valid for the networks the embedded targets will be plugged into. You may have to check with a system administrator to obtain an available IP address.



Completing the Rest of the Application Startup and Initialization

The remaining part of NppCreateMultitasking creates the queue and the tasks.

```
void
NppCreateMultitasking( void* first_available_memory )
{
    /***** Nucleus C++ components *****/
    ... Nucleus C++ component initialization is described above.

    /***** Demonstration application *****/

    /* Create TCP_Server_Task. */
    status = NU_Create_Task(&TCP_Server_Task_CB, "tcp_serv",
        TCP_Server_Task, 0, NU_NULL, new char[TASK_STACK_SIZE],
        TASK_STACK_SIZE, NORMAL_TASK_PRIORITY, 0,
        NU_PREEMPT, NU_START);
    if (status != NU_SUCCESS)
    {
        error("Cannot create TCP_Server_Task.");
    }

    /* Create TCP_Echo_Task. */
    status = NU_Create_Task(&TCP_Echo0_Task_CB, "tcp_echo", TCP_Echo_Task,
        0, NU_NULL, new char[TASK_STACK_SIZE], TASK_STACK_SIZE,
        NORMAL_TASK_PRIORITY, 0, NU_PREEMPT, NU_START);
    if (status != NU_SUCCESS)
    {
        error("Cannot create TCP_Echo_Task.");
    }

    /* Create TCP_Client_Task. */
    status = NU_Create_Task(&TCP_Client_Task_CB, "tcp_cli",
        TCP_Client_Task, 0, NU_NULL, new char[TASK_STACK_SIZE],
        TASK_STACK_SIZE, NORMAL_TASK_PRIORITY, 0, NU_PREEMPT,
        NU_NO_START);
    if (status != NU_SUCCESS)
    {
        error("Cannot create TCP_Client_Task.");
    }

    /* Create UDP_Server_Task. */
    status = NU_Create_Task(&UDP_Server_Task_CB, "udp_serv",
        UDP_Server_Task, 0, NU_NULL, new char[TASK_STACK_SIZE],
        TASK_STACK_SIZE, NORMAL_TASK_PRIORITY, 0, NU_PREEMPT,
        NU_START);
    if (status != NU_SUCCESS)
    {
        error("Cannot create UDP_Server_Task.");
    }

    /* Create UDP_Client_Task. */
    status = NU_Create_Task(&UDP_Client_Task_CB, "udp_cli",
        UDP_Client_Task, 0, NU_NULL, new char[TASK_STACK_SIZE],
        TASK_STACK_SIZE, NORMAL_TASK_PRIORITY, 0, NU_PREEMPT,
        NU_NO_START);
}
```



```

if (status != NU_SUCCESS)
{
    error("Cannot create UDP_Client_Task.");
}

/* Create Comm_Task */
status = NU_Create_Task(&Comm_Task_CB, "comm_task", Comm_Task, 0,
    NU_NULL, new char[TASK_STACK_SIZE], TASK_STACK_SIZE,
    COMM_TASK_PRIORITY, 0, NU_PREEMPT, NU_START);
if (status != NU_SUCCESS)
{
    error("Cannot create Comm_Task.");
}

/* Create Socket queue. */
status = NU_Create_Queue(&socketQueue, "SocQueue",
    new char[2*sizeof(NuTcpSocketPointer)], 2, NU_FIXED_SIZE,
    1, NU_FIFO);
if (status != NU_SUCCESS)
{
    error("Cannot create Socket Queue.");
}

#if (INCLUDE_LOOPBACK_DEVICE == NU_TRUE)
/* Create TCP_Loopback_Client_Task. */
status = NU_Create_Task(&TCP_Loopback_Client_Task_CB, "lb_cli",
    TCP_Loopback_Client_Task, 0, NU_NULL,
    new char[TASK_STACK_SIZE], TASK_STACK_SIZE,
    NORMAL_TASK_PRIORITY, 0, NU_PREEMPT, NU_START);
if (status != NU_SUCCESS)
{
    error("Cannot create TCP_Loopback_Client_Task.");
}

/* Create TCP_Echo_Task. We will need a second one to handle
the loopback client. */
status = NU_Create_Task(&TCP_Echo1_Task_CB, "tcp_echo",
    TCP_Echo_Task, 0, NU_NULL, new char[TASK_STACK_SIZE],
    TASK_STACK_SIZE, NORMAL_TASK_PRIORITY, 0, NU_PREEMPT,
    NU_START);
if (status != NU_SUCCESS)
{
    error("Cannot create TCP_Echo_Task.");
}
#endif
}

```

Application Tasks

The various application tasks are identical in pattern to the application tasks found within the “C” version of the NETDEMO application. However, the networking portions of the tasks are object-oriented and use the Nucleus C++ NET socket classes.



TCP Server Task

The TCP server task handles host-side Winsock client connections. It is found within the NPP\NPPNET\DEMO\SIMPLE\NETD.CPP file.

```
VOID TCP_Server_Task(UNSIGNED argc, VOID *argv)
```

It follows the following pseudo-code pattern:

- Create a TCP socket object.
- Bind the socket to the device's IP address, port 7 and set it up to accept up to 10 simultaneous client connections.
- Loop forever.
 - Suspend on the socket until a client connects.
 - Check the validity of the worker socket associated with the new client connection and pass it to the TCP echo server worker task to handle it.

Creating the TCP server socket object is accomplished by constructing a NuTcpSocket object and invoking its Create member.

```
// Create the TCP server socket.
NuTcpSocket server_socket;
if( !server_socket.Create() )
{
    error("Cannot create.");
}
```

Binding to the device's IP address and port 7 is accomplished by constructing a SocketAddress object to encapsulate the device's socket address and passing it as a parameter into the TCP server socket's Bind member.

```
// Bind to the device IP address and port 7.
IPAddress server_ip_address( (UINT32)IP_ADDR_ANY );
SocketAddress server_address( server_ip_address, 7, "TCP_Echo" );
if( server_socket.Bind( server_address ) < 0 )
{
    error("Cannot bind.");
}
```

Getting ready to accept up to 10 simultaneous client connections is accomplished by executing the Listen member on the TCP server socket.

```
// Be ready to accept 10 connection requests.
if( server_socket.Listen( 10 ) != NU_SUCCESS )
{
    error("Cannot listen.");
}
```



Constructing a worker socket to handle the next connection is accomplished by creating a new `NuTcpSocket` object. We are creating the worker socket on the heap since the lifetime of the socket is managed by the worker task. The worker task deletes the object when it completes the handling of the specific client connection.

Note that you do not have to invoke the `Create` member on the new worker socket in this case because the network stack does it for you in the `Accept` member (see below).

```
// Construct a new "worker" socket to handle this connection.
NuTcpSocket* worker_socket = new NuTcpSocket;
```

Suspending until a client connects is accomplished by invoking the `Accept` member on the TCP server socket with the worker socket passed in as a parameter and another temporary `SocketAddress` object passed in to hold the connecting client's IP address and port number. The client's `SocketAddress` object is filled in by `Accept`. It is not used in this example.

```
// Call Accept on the server socket to accept a new client connection.
SocketAddress client_address;
if( server_socket.Accept( worker_socket, &client_address ) < 0 )
{
    error("Cannot accept.");
}
```

Checking the validity of the worker socket is easy since the socket class has a specific member for this named `IsValid`.

```
// Make sure the worker socket is valid.
if( !worker_socket->IsValid() )
{
    error("Worker socket not valid.");
}
```

Passing the worker socket to the TCP echo server worker task is performed by sending its pointer to the global inter-task communication queue. The TCP echo server worker task blocks on this queue until a pointer to a socket is sent to it.

```
// Send a pointer to the worker socket to the echo task.
NU_Send_To_Queue
(
    &socketQueue, (NuTcpSocketPointer*)&worker_socket, 1, NU_SUSPEND
);
```

After the worker socket associated with the connection is sent to the TCP echo worker task, the TCP server task loops and creates another worker socket for the next connection. It then blocks again on another call to `Accept` waiting for that connection.

For each client that connects, a new worker socket is associated with the connection and the socket is passed to the single instance of the echo worker task. This simple demo only has one worker thread but of course you can create a separate worker thread to handle each connection concurrently.



TCP Echo Server Worker Task

The TCP echo server worker thread handles the host-side Winsock client connection. It is found within the `NPP\NPPNET\DEMO\SIMPLE\NETD.CPP` file.

```
VOID TCP_Echo_Task(UNSIGNED argc, VOID *argv)
```

It follows the following pseudo-code pattern:

- Loops forever.
 - Suspends until a pointer to a TCP socket associated with a client connection is sent to the queue.
 - Loop while connected to the client.
 - Turn *on* blocking during the receive operation, receive data from the client, and then turn blocking *off*.
 - Echo the data back to the client.
 - Close the TCP socket.
 - Delete the TCP socket to return its resources.

Blocking on the queue until a socket pointer is sent by the TCP server task is accomplished by using the Nucleus PLUS queue receive service, specifying suspension until a message is received.

```
// Retrieve the worker socket from the socket queue.
NuTcpSocket* worker_socket;
UNSIGNED actSize;
if
(
    NU_Receive_From_Queue
    (
        &socketQueue, (NuTcpSocketPointer*)&worker_socket, 1, &actSize,
        NU_SUSPEND
    )
    !=
    NU_SUCCESS
)
{
    error("Unable to receive socket from queue.");
}
```

Turning on blocking during the receive operation is performed by simply invoking the `SetBlocking` method on the worker socket object, passing in `TRUE` to turn blocking on. Turning blocking off is performed by invoking the same `SetBlocking` method accept that `FALSE` is specified.

```
// Turn on the "block during a read" flag.
worker_socket->SetBlocking( TRUE );
```



Receiving data from the client is as simple as executing the `Recv` operation on the worker socket associated with the client connection.

```
// Receive some data from the client, blocking until it is received.
INT32 bytes_received = worker_socket->Recv
(
    tcp_rx_buffer, (UINT16)TCP_SERVER_MAX_RX_SIZE
);
```

To check the results of the receive operation, examine the return value of the `Recv` method. If it is `NU_NOT_CONNECTED`, the client is no longer connected. If the return value is negative, an error has occurred.

```
// Check results.
if( bytes_received == NU_NOT_CONNECTED )
{
    // No longer connected.
    connected = FALSE;
}
else if( bytes_received < 0 )
{
    error("Cannot receive.");
}
```

Echoing the data back to the client is as simple as invoking the `Send` member on the worker socket associated with the client with the same data that was received.

```
// Successful receive data, echo it back to the client.
INT32 bytes_sent = worker_socket->Send
(
    tcp_rx_buffer, (UINT16)bytes_received
);
```

To check the results of the send operation, examine the return value of the `Send` method. If it is `NU_NOT_CONNECTED`, the client is no longer connected. If the return value is anything other than the number of bytes you requested to send, an error has occurred.

```
// Check results.
if( bytes_sent != bytes_received )
{
    // Could not send the data, check if it is due to a lost connection.
    if( bytes_sent == NU_NOT_CONNECTED )
    {
        // No longer connected.
        connected = FALSE;
    }
    else
    {
        error("Cannot send.");
    }
}
```



To close the socket, simply invoke the `Close` member on it.

```
// Close the connection.
if( worker_socket->Close() != NU_SUCCESS )
{
    error("Cannot close.");
}
```

To destroy the socket and return its resources to the system, simply call `delete` on the pointer to the worker socket object. The object was newed in the TCP server task.

```
// Delete the worker socket.
delete worker_socket;
```

TCP Client Task

The TCP client thread connects to the host-side Winsock server. It is found within the `NPP\NPPNET\DEMO\SIMPLE\NETD.CPP` file.

```
VOID TCP_Client_Task(UNSIGNED argc, VOID *argv)
```

It follows the following pseudo-code pattern:

- Create a TCP socket object.
- Connect it to the server.
- Disable the Nagle Algorithm for the socket.
- Loop for the number of times the host requested.
 - Send data to the server.
 - Turn *on* blocking during the receive operation, receive all of the echoed data from the server, and then turn blocking *off*.
- Close the socket.

Creating the TCP client socket object is accomplished by constructing a `NuTcpSocket` object and invoking its `Create` member.

```
// Create the TCP client socket.
NuTcpSocket client_socket;
if( !client_socket.Create() )
{
    error("Cannot create.");
}
```



To build a socket address object that encapsulates the server's address, the demo constructs a `SocketAddress` object using the global `host_address` as a constructor parameter. `host_address` is a `SocketAddress` object that has been filled in with the host-side socket address' information by the communication task when the host-side NETDEMO application connected.

```
// Create a socket address object with host_address's attributes, port 7.
SocketAddress server_address( host_address );
server_address.SetPort( 7 );
server_address.SetName( "TCP_Echo" );
```

Once we have a `SocketAddress` object that encapsulates the server's address, we can connect to it by invoking the `Connect` method on the client socket object using it as the server address parameter.

```
// Connect to the server.
if( client_socket.Connect( server_address ) < 0 )
{
    error("Cannot connect.");
}
```

To disable the Nagle Algorithm on the client socket, call its `EnableTcpDelay` member with a `FALSE` parameter. The Nagle Algorithm is a means for coalescing many small packets into a single large packet. It delays the transmittal of a small data packet for a brief (1/4 second by default) period of time, in the expectation that the application will have another small packet to send which can be combined with the current one.

```
// Disable the Nagle Algorithm.
client_socket.EnableTcpDelay( FALSE );
```

Sending and receiving data to and from the server and checking the results is accomplished in the same manner as the TCP echo server did as described above. The only difference is that the client sends data to the server first, which echoes it back to the client. The client then receives after it sends in order to catch the echoed data.

```
// Send the data.
INT32 bytes_sent = client_socket.Send( tcp_tx_buffer, TCP_Byte );
```

You will note that it is possible that not all of the echoed data will be received in a single receive call. The TCP client task has a slightly different receive algorithm that accumulates the number of bytes received until it reaches the number sent. It then loops again to send more data.

```
// Get the server's response.
bytes_received = client_socket.Recv( tcp_rx_buffer, TCP_Byte );
```



To close the socket, simply invoke the `Close` member on it.

```
// Close the connection.
if( client_socket.Close() != NU_SUCCESS )
{
    error("Cannot close.");
}
```

UDP Server Task

The UDP server task handles host-side Winsock client connections. It is found within the `NPP\NPPNET\DEMO\SIMPLE\NETD.CPP` file.

```
VOID UDP_Server_Task(UNSIGNED argc, VOID *argv)
```

It follows the following pseudo-code pattern:

- Create a UDP socket object.
- Bind the socket to the device's IP address, port 7.
- Loop forever.
 - Receive data from the client.
 - Echo the data back to the client.

Creating the UDP server socket object is accomplished by constructing a `NuUdpSocket` object and invoking its `Create` member.

```
// Create the UDP server socket.
NuUdpSocket server_socket;
if( !server_socket.Create() )
{
    error("Cannot create.");
}
```

Binding to the device's IP address and port 7 is accomplished by constructing a `SocketAddress` object to encapsulate the device's socket address and passing it as a parameter into the UDP server socket's `Bind` member.

```
// Bind to the device IP address and port 7.
IPAddress server_ip_address( (UINT32)IP_ADDR_ANY );
SocketAddress server_address( server_ip_address, 7, "UDP_Echo" );
if( server_socket.Bind( server_address ) < 0 )
{
    error("Cannot bind.");
}
```



Receiving data from the client is as simple as executing the `RecvFrom` operation on the server socket. In the following code snippet, `client_address` is a `SocketAddress` object that the `RecvFrom` method fills in with the socket address information of the client that sent the data that is received.

```
// The client's address.
SocketAddress client_address;

// Receive data from the client.
INT32 bytes_received = server_socket.RecvFrom
(
    udp_rx_buffer, UDP_SERVER_MAX_RX_SIZE, client_address
);
```

If the return value if the `RecvFrom` member is negative, an error has occurred.

```
// Check results.
if( bytes_received < 0 )
{
    error("Cannot receive.");
}
```

To send the data using UDP, the `SendTo` member of the server socket object is used. The client address socket object that was filled in by the `RecvFrom` member is used as the destination address.

```
// Echo the data back to the client.
INT32 bytes_sent = server_socket.SendTo
(
    udp_rx_buffer, (UINT16)bytes_received, client_address
);
```

If the `SendTo` member returns a negative value, an error has occurred.

```
// Check results.
if( bytes_sent < 0 )
{
    error("Cannot send.");
}
```

UDP Client Task

The UDP client thread connects to the host-side Winsock server. It is found within the `NPP\NPPNET\DEMO\SIMPLE\NETD.CPP` file.

```
VOID UDP_Client_Task(UNSIGNED argc, VOID *argv)
```



It follows the following pseudo-code pattern:

- Create a UDP socket object.
- Loop for the number of times the host requested.
 - Send data to the server.
 - Receive all of the echoed data from the server.
- Send the quit command to the server.
- Close the socket.

Creating the UDP client socket object is accomplished by constructing a NuUdpSocket object and invoking its Create member.

```
// Create the UDP client socket.
NuUdpSocket client_socket;
if( !client_socket.Create() )
{
    error("Cannot create.");
}
```

Before data can be sent to the server, we must build a socket address object with the host's address in the same manner as we did in the TCP client task.

```
// Create a socket address object with host_address's attributes, port 7.
SocketAddress server_address( host_address );
server_address.SetPort( 7 );
server_address.SetName( "UDP_Echo" );
```

Sending data and checking results is done the same way that the server sends data.

```
// Send data to server.
INT32 bytes_sent = client_socket.SendTo
(
    udp_tx_buffer, UDP_Byte, server_address
);
```

Receiving data and checking results is done the same way that the server receives data.

```
// Get data from server.
INT32 bytes_received = client_socket.RecvFrom
(
    udp_rx_buffer, UDP_Byte, server_address
);
```

Sending the quit command to the server is accomplished by sending it the letter 'q'.

```
// Send quit command to the server.
INT32 bytes_sent = client_socket.SendTo( "q", 1, server_address );
```



To close the socket, simply invoke the `Close` member on it.

```
// Close the connection.
if( client_socket.Close() != NU_SUCCESS )
{
    error("Cannot close.");
}
```

TCP Client Loopback Task

The TCP client loopback thread connects to the TCP server task on hardware reference platforms that do not have a real network driver. The application is very similar to the TCP client task described above and the shared details are not repeated here.

```
VOID TCP_Loopback_Client_Task(UNSIGNED argc, VOID *argv)
```

The only difference between the loopback version is the socket address of the server and a few Nucleus PLUS relinquish service calls to prevent task starvation since the loopback device is very fast. The socket address of the server is the socket address of the device since we will be connecting into the TCP server task that has been bound to the device's IP address within this same application.

Host-Side Communications/Command Task

The host-side communications task communicates to the host-side Winsock application using both UDP and TCP to accept commands that control the demo.

We will not repeat the same information described above on how this task creates sockets, binds to the devices, accepts connections, and sends and receives data since it is identical in principle to the way the other tasks in the system do this. We will however highlight a few differences that this task has that demonstrates typical situations in real-world networking applications.

The main difference is that the communications thread handles more than one socket. This demonstrates how to suspend waiting for network connections on multiple sockets. To do this, the task first creates a `FD_SET` structure and initializes it.

```
// Wait for data on either socket.
FD_SET readfs;
NuSocket::FD_Init( &readfs );
```

It then operates on the sockets that the thread will eventually block on using the `FD_Set` method of the socket classes. This sets up the `FD_SET` structure for a call to `NU_Select`.

```
// Set the bits corresponding to the two sockets.
tcp_server_socket.FD_Set( &readfs );
udp_server_socket.FD_Set( &readfs );
```



The communications task then calls `NU_Select` which suspends the thread until a client connects to either of the two sockets. After `NU_Select` returns with `NU_SUCCESS` (indicating a client has connected), the task checks which of the two sockets has the connection by invoking the `FD_Check` member on each of the socket instances.

```
// Suspend waiting for data on either socket.
if( NU_Select( NSOCKETS, &readfs, NULL, NULL, NU_SUSPEND ) == NU_SUCCESS )
{
    // Data received on at least one of the sockets...

    if( udp_server_socket.FD_Check( &readfs ) )
    {
        // Yes, there is data on the udp socket...
    }

    if( tcp_server_socket.FD_Check( &readfs ) )
    {
        // Yes, there is data on the tcp socket...
    }
}
```

The rest of the details of the communication task are very algorithmic and do not aide in the understanding of Nucleus C++ NET.



2

Nucleus C++ NET Class Reference



class IPAddress

This class is an object interface for a basic 32 bit IP address. It contains a number of different constructors and access members as well as casting operators to ease use. It contains as a data member the Nucleus NET C data structure `struct id_struct`, a basic encapsulation of a 32 bit IP address. Here is the "C" definition of the `struct id_struct`. This can be found in the Nucleus NET file "SOCKETD.H".

```
// 32-bit structure containing 4-byte IP address number.
struct id_struct
{
    UINT8 is_ip_addrs[4];
};
```

Public Member Functions

Member	Overview
<code>~IPAddress</code>	Destructor
<code>Get</code>	Called to retrieve the IP address as a 32 bit number.
<code>IPAddress</code>	Constructor utilizing individual octets.
<code>IPAddress</code>	Constructor
<code>IPAddress</code>	Constructor using 32 bit IP address.
<code>IPAddress</code>	Copy constructor for "C" structure.
<code>IsClassA</code>	Determines if this is a class A IP address.
<code>IsClassB</code>	Determines if this is a class B IP address.
<code>IsClassC</code>	Determines if this is A class C IP address.
<code>IsClassD</code>	Determines if this IP address belongs to class D.
<code>MaskClassA</code>	Called to mask bits necessary to make this a class A address.
<code>MaskClassB</code>	Called to mask bits necessary to make this a class B address.
<code>MaskClassC</code>	Called to mask bits necessary to make this a class C address.
<code>MaskClassD</code>	Called to mask bits necessary to make this a class D address.
<code>operator &=</code>	Bit-wise AND assignment operator.
<code>operator []</code>	Access to individual octets (bytes) within the IP address.
<code>operator []</code>	Access to individual octets.
<code>operator =</code>	Bitwise OR assignment operator.
<code>operator =</code>	Assignment operator.
<code>operator =</code>	Assignment operator from "C" IP address.
<code>operator const struct id_struct*()</code>	Casting operator that extracts a const pointer to the underlying C structure.
<code>Set</code>	IP address set by octets (individual bytes).
<code>Set</code>	32 bit Set.



Protected Member Functions

Member	Overview
<code>operator struct id_struct*</code>	Casting operator for access to the raw IP address.

IPAddress::~IPAddress

```
virtual  
IPAddress::~IPAddress();
```

Destructor

Overview

Condition	Description
Pre-condition	None
Action	None
Post-condition	The object does not exist.

Parameters

None

Return Value

None

Example

Destructors are called automatically when the object goes out of scope or is deleted.



IPAddress::Get

```
inline
UINT32
IPAddress::Get ()
const;
```

Called to retrieve the IP address as a 32 bit value.

Overview

Condition	Descriptor
Pre-condition	None
Action	Returns the raw 32 bit IP address.
Post-condition	The IP address is returned. The IP Address object is unmodified.

Parameters

None

Return Value

Return Value	Overview
validData	the UINT32 value contains the IP address stored in the IPAddress object.

Example

```
const IPAddress ip( 200, 100, 50, 1 );
UINT32 theip;

theip = ip.Get();
```



IPAddress::IPAddress

```
IPAddress::IPAddress  
(  
const struct id_struct& newId  
);
```

Copy Constructor for the IPAddress object that uses a Nucleus NET C structure for initialization.

Overview

Conditon	Description
Pre-condition	'newId' - is a struct id_struct structure, or a reference to one.
Action	Initializes the IPAddress object to the identical IP address specified by newId.
Post-condition	The object exists and is identical to that specified by newId.

Parameters

Parameter	Overview
newId	copy Constructor

Return Value

None

Example

```
struct id_struct ip = { 200, 100, 50, 1 };  
  
IPAddress myip2( ip ); // copy constructor...
```



IPAddress::IPAddress

```
IPAddress::IPAddress  
(  
    UINT8 u1,  
    UINT8 u2,  
    UINT8 u3,  
    UINT8 u4  
);
```

Constructor. Builds an IP address from individual octets.

Overview

Condition	Description
Pre-condition	None
Action	Initializes the IPAddress object using the given bytes. u1 is the most significant byte, u4 is the least significant byte.
Post-condition	The object exists and has been initialized with the given data.

Parameters

Parameter	Overview
u1	The most significant byte of the IP address.
u2	The second byte of the IP address.
u3	The third byte of the IP address.
u4	The least significant byte of the IP address.

Return Value

None

Example

```
IPAddress ip1( 200, 100, 50, 1 );
```



IPAddress::IPAddress

```
IPAddress::IPAddress
(
    UINT32 u32IP
);
```

Constructor. Builds an IP address from a 32 bit unsigned integer. This is the default constructor.

Overview

Condition	Description
Pre-condition	None
Action	Initializes the IPAddress object using the given integer.
Post-condition	The object exists and has been initialized with the given data.

Parameters

Parameter	Overview
u32IP	Contains the desired data. Default value is 0. (0.0.0.0)

Return Value

None

Example

```
IPAddress ip( 0x12345678 ); // By 32 bit value.

IPAddress ip2; // Default constructor.
```



IPAddress::IPAddress

```
IPAddress::IPAddress  
(  
const IPAddress& newIP  
);
```

Constructor. Copy constructor for the IPAddress object.

Overview

Condition	Description
Pre-condition	'newIP' - is an IPAddress object, or reference to an IPAddress object.
Action	Initializes the IPAddress object to the identical IP address specified by newIP.
Post-condition	This IP address object is identical to that specified by newIP.

Parameters

Parameter	Overview
newIP	Copy Constructor.

Return Value

None

Example

```
IPAddress ip1( 200, 100, 50, 1 );  
  
IPAddress ip2( ip1 ); // Copy constructor.
```



IPAddress::IsClassA

```
inline  
BOOL  
IPAddress::IsClassA()  
const;
```

Called to test if this IP address is a "class A" address.

Overview

Condition	Description
Pre-condition	None
Action	Tests highest order bit (31) to see if it is 0.
Post-condition	None

Parameters

None

Return Value

Return Value	Overview
FALSE	This is not a class A address.
TRUE	This is a class A IP address.

Example

```
IPAddress ip( 200, 100, 50, 0 ); // A class A address.  
  
if( ip.IsClassA() )  
{  
    ...  
}
```



IPAddress::IsClassB

```
inline  
BOOL  
IPAddress::IsClassB()  
const;
```

Called to test if this IP address is a "class B" address.

Overview

Condition	Description
Pre-condition	None
Action	Tests if highest order bits (31- 30) are 1 and 0 respectively.
Post-condition	None

Parameters

None

Return Value

Return Value	Overview
FALSE	This is not a class B address.
TRUE	This is a class B IP address.

Example

```
IPAddress ip( 200, 100, 50, 0 ); // A class B address.  
  
if( ip.IsClassB() )  
{  
    ...  
}
```



IPAddress::IsClassC

```
inline  
BOOL  
IPAddress::IsClassC()  
const;
```

Called to test if this IP address is a "class C" address.

Overview

Condition	Description
Pre-condition	None
Action	Tests highest order bits (31-29) to see if they are 110 respectively.
Post-condition	None

Parameters

None

Return Value

Return Value	Overview
FALSE	This is not a class C address.
TRUE	This is a class C IP address.

Example

```
IPAddress ip( 200, 100, 50, 1 );  
  
if( ip.IsClassC() )  
{  
    ...  
}
```



IPAddress::IsClassD

```
inline  
BOOL  
IPAddress::IsClassD()  
const;
```

Called to test if this IP address is a "class D" address. Class D addresses are used for multicasting.

Overview

Condition	Description
Pre-condition	None
Action	Tests highest order bits (31-28) to see if they are 1110 respectively.
Post-condition	None

Parameters

None

Return Value

Return Value	Overview
FALSE	This is not a class D (multicast) address.
TRUE	This is a class D IP address.

Example

```
// All systems on a LAN multicast address (class D).  
IPAddress ip( 200,100, 50, 0 );  
  
if( ip.IsClassD() )  
{  
    ...  
}
```



IPAddress::MaskClassA

```
inline  
VOID  
IPAddress::MaskClassA();
```

Called to mask bits necessary to make this a class A address.

Overview

Condition	Description
Pre-condition	None
Action	Sets bit 31 to 0.
Post-condition	The IP address is now a class A IP address.

Parameters

None

Return Value

Return Value	Overview
VOID	Nothing returned.

Example

```
IPAddress ip;  
...  
ip.MaskClassA(); // Sets bit 31 to 0.
```



IPAddress::MaskClassB

```
inline  
VOID  
IPAddress::MaskClassB();
```

Called to mask bits necessary to make this a class B address.

Overview

Condition	Description
Pre-condition	None
Action	Set highest order bits (31-30) to 10 respectively.
Post-condition	This is a class B IP address.

Parameters

None

Return Value

Return Value	Overview
VOID	Nothing returned.

Example

```
IPAddress ip;  
...  
ip.MaskClassB(); // Sets bit 31-30 to 10.
```



IPAddress::MaskClassC

```
inline  
VOID  
IPAddress::MaskClassC();
```

Called to mask bits necessary to make this a class C address.

Overview

Condition	Description
Pre-condition	None
Action	Set highest order bits (31-29) to 110 respectively.
Post-condition	This is now a class C IP address.

Parameters

None

Return Value

Return Value	Overview
VOID	Nothing returned.

Example

```
IPAddress ip;  
  
ip.MaskClassC(); // Set bits 31-29 to 110.
```



IPAddress::MaskClassD

```
inline  
VOID  
IPAddress::MaskClassD();
```

Called to mask bits necessary to make this a class D address.

Overview

Condition	Description
Pre-condition	None
Action	Set highest order bits (31-28) to 1110 respectively.
Post-condition	This is now a class D (multicast) IP address.

Parameters

None

Return Value

Return Value	Overview
VOID	Nothing returned.

Example

```
IPAddress mask( 0 );  
mask.MaskClassD(); // Mask is now 0xE0000000.
```



IPAddress::operator &=

```
inline
IPAddress&
IPAddress::operator &=
(
const IPAddress& mask
);
```

Bitwise AND assignment operator. This can be used for masking an IP address with a given subnet mask.

Overview

Condition	Description
Pre-condition	None
Action	AND's this instance with that specified by mask and stores the result in this instance.
Post-condition	This object's data has been bitwise AND'ed with mask. The passed in mask IP address object is unmodified.

Parameters

Parameter	Overview
mask	A reference to an IPAddress object.

Return Value

Return Value	Overview
aValidReference	A reference to the current object.

Example

```
IPAddress subnetmask( 200, 100, 50, 0 );
IPAddress destination( 200, 100, 50, 0 );
...
// Figure out the gateway address for the given IP
// address by anding with the subnetmask.

destination &= subnetmask;
```



IPAddress::operator []

```
inline
UINT8
IPAddress::operator []
(
    INT nIndex
)
const;
```

Operator for access to the individual octets (bytes) within the IPAddress. This version does not allow modification of the object, but simply returns a copy of the byte of interest.

Overview

Condition	Description
Pre-condition	The object exists.
Action	Returns a copy of the desired byte.
Post-condition	None

Parameters

Parameter	Overview
nIndex	Contains an offset into the IPAddress object, valid values are from 0 to 3, 0 being the Most significant byte (class A) and 3 being the least significant byte (class D). caution: there is no error checking done on this

Return Value

Return Value	Overview
validData	A copy of the given byte is returned.

Example

```
const IPAddress  myip( 200, 100, 50, 0 );
UINT8 temp;

temp = myip[3]; // Gets the 17 value.

myip[2] = temp; // Compiler error, can't modify a const object.
```



IPAddress::operator []

```
inline
UINT8&
IPAddress::operator []
(
    INT nIndex
);
```

Operator for access to the individual octets (bytes) within the IPAddress. This version is used to modify the IPAddress object.

Overview

Condition	Description
Pre-condition	The object exists.
Action	Returns a reference to the desired byte (allowing modification).
Post-condition	The byte specified by the given index has been set to the passed in value.

Parameters

Parameter	Overview
nIndex	Contains an offset into the IPAddress object, valid values are from 0 to 3, 0 being the Most significant byte and 3 being the least significant byte. Caution: there is no error checking done on this parameter.

Return Value

Return Value	Overview
aValidReference	A reference to the byte specified by the index. This allows direct modification of the individual byte.

Example

```
IPAddress myip( 200, 100, 50, 0 );
UINT8    temp;

temp = myip[3]; // Gets the 17 value (const).
myip[2] = temp; // Sets the 38 value (non-const).
```



IPAddress::operator |=

```
inline  
IPAddress&  
IPAddress::operator |=  
(  
const IPAddress& mask  
);
```

Bitwise OR assignment operator. This can be used for masking an IP address with a desired mask.

Overview

Condition	Description
Pre-condition	The object exists.
Action	OR's this instance with that specified by mask and stores the result in this instance.
Post-condition	This object's data has been bitwise OR'ed with mask. The mask parameter is unchanged.

Parameters

Parameter	Overview
mask	The IP address to bitwise OR with.

Return Value

Return Value	Overview
aValidReference	A reference to the current object.

Example

```
IPAddress mask( 0x80, 0, 0, 0 );  
IPAddress destination( 200, 100, 50, 0 );  
...  
// Manually set the most significant bit by OR'ing with mask.  
destination |= mask;
```



IPAddress::operator =

```
inline
IPAddress&
IPAddress::operator =
(
const struct id_struct& ids
);
```

Assignment operator given a "C" id_struct structure.

Overview

Condition	Description
Pre-condition	The object exists.
Action	Calls set member.
Post-condition	This object has been modified.

Parameters

Parameter	Overview
ids	A reference to the Nucleus NET structure id_struct. This is a simple structure that contains an array of 4 unsigned characters.

Return Value

Return Value	Overview
aValidReference	A reference to the current object.

Example

```
UINT8 bob[] = { 200, 100, 50, 0 };

IPAddress ip;

// Use the assignment operator by casting the C array
// (since that is exactly what id_struct is anyway).
ip = (struct id_struct)bob;
```



IPAddress::operator =

```
inline  
IPAddress&  
IPAddress::operator =  
(  
const IPAddress& newip  
);
```

Assignment operator for copying the contents of one IPAddress object to another.

Overview

Condition	Description
Pre-condition	The object exists.
Action	Calls set member.
Post-condition	This object has been changed. The passed in IP address object is unmodified.

Parameters

Parameter	Overview
newip	The IP address to copy.

Return Value

Return Value	Overview
aValidReference	A reference to the current object.

Example

```
IPAddress ip1( 0x12345678 );  
IPAddress ip2;  
Ip2 = ip1; // Assignment operator.
```



IPAddress::operator const struct id_struct*()

```
inline
struct id_struct*
IPAddress::operator const struct id_struct*() ()
const;
```

Const casting operator that allows you to use IPAddress objects in place of const pointers to C IP addresses. This is mainly for compatibility with Nucleus NET C API functions.

Overview

Condition	Description
Pre-condition	None
Action	Returns a const pointer to the internal struct id_struct member.
Post-condition	None

Parameters

None

Return Value

Return Value	Overview
aValidPointer	A pointer to the internal C id_struct structure is returned. It is not possible for it to be NULL provided that this object exists.

Example

```
const IPAddress the_empire( 200, 100, 50, 0 );

UINT8 ip[4];

// Manually copy the underlying ip address into a memory
location.
memcpy( &ip, (const struct id_struct*)the_empire, 4 );
```



IPAddress::operator struct id_struct*

```
inline  
struct id_struct*  
IPAddress::operator struct id_struct*();
```

Casting operator that allows you to use IPAddress objects in place of pointers to C IP addresses. This is mainly for compatibility with Nucleus NET C API functions.

Overview

Condition	Description
Pre-condition	None
Action	Returns a pointer to the internal struct id_struct member.
Post-condition	None

Parameters

None

Return Value

Return Value	Overview
aValidPointer	A pointer to the internal C id_struct structure is returned. It is not possible for it to be NULL provided that this object exists.

Example

```
IPAddress subnet( 255, 255, 255, 50 );  
  
// Extract a pointer to the structure.  
struct id_struct* pStruct = (struct id_struct*)subnet;
```



IPAddress::Set

```

inline
VOID
IPAddress::Set
(
    UINT8 u1,
    UINT8 u2,
    UINT8 u3,
    UINT8 u4
);

```

Sets the object's IP address by individual octets (bytes).

Overview

Condition	Description
Pre-condition	None
Action	Assigns each byte individually.
Post-condition	The IP address has been set.

Parameters

Parameter	Overview
u1	Contains most significant byte of the IP address.
u2	Contains the second byte of the IP address.
u3	Contains the third byte of the IP address.
u4	Contains the least significant byte of the IP address.

Return Value

Return Value	Overview
VOID	Nothing returned.

Example

```

IPAddress myip;
myip.Set( 200, 100, 50, 1 );

```



IPAddress::Set

```
inline  
VOID*  
IPAddress::Set  
(  
    UINT32 u32IP  
);
```

Sets the objects IP address via a 32 bit number.

Overview

Condition	Description
Pre-condition	None
Action	Sets the internal IP address via a 32 bit assignment.
Post-condition	The IP address has been set.

Parameters

Parameter	Overview
u32IP	Contains the desired bytes for the IP address.

Return Value

Return Value	Overview
VOID	Nothing returned.

Example

```
IPAddress mask;  
mask.Set( 0xFFFFFFFF );
```



class NppNET : public NppComponent

The Nucleus C++ NET class.

Public Member Functions

Member	Overview
~NppNET	Destructor
Initialize	Final initialize occurs after the object completely exists.
NppNET	Constructor
RegisterDevice	Registers an instance of an NuNetDevice derived object with the network stack.
UnRegisterDevice	In the future, this member will be used for removal of devices. Nucleus NET does not currently support removal of devices.

Protected Member Functions

Member	Overview
InitializeClasses	Internal member to initialize static class members.
InitializeNetwork	Initializes the networking stack.



NppNET : : ~NppNET

```
virtual  
VOID*  
NppNET : : ~NppNET ( ) ;
```

Destructor

Overview

Condition	Description
Pre-condition	None
Action	None
Post-condition	The object does not exist.

Parameters

None

Return Value

None

Example

Destructors are called automatically when the object goes out of scope or is deleted.



NppNET::Initialize

```
virtual
STATUS
NppNET::Initialize();
```

Final initialization occurs after the object completely exists.

Overview

Condition	Description
Pre-condition	The NppBASE component has been initialized.
Action	Initializes the underlying Nucleus NET networking stack as well as static portions of Nucleus C++ NET classes.
Post-condition	The NppNET object is initialized. The network stack is ready for devices to be registered.

Parameters

None

Return Value

Return Value	Overview
NU_SUCCESS	Indicates successful completion of the service.
value < 0	The operation failed and an error code was returned.

Example

```
NppNET* pNetworkComponent = new NppNET;
pNetworkComponent->Initialize();
```



NppNET::InitializeClasses

```
virtual  
STATUS*  
NppNET::InitializeClasses();
```

Called by the NppNET component to initialize static portions of Nucleus C++ NET classes.

Overview

Condition	Description
Pre-condition	None
Action	Initializes static portions of classes that have them.
Post-condition	The classes in Nucleus C++ NET are ready to be used.

Parameters

None

Return Value

Return Value	Overview
NU_SUCCESS	Indicates successful completion of the service.
NU_UNAVAILABLE	The member failed because a needed resource was unavailable.
value <0	The operation failed and an error code was returned.

Example

```
NppNET::Initialize()  
{  
    InitializeClasses();  
    ...  
}
```



NppNET::InitializeNetwork

```
virtual
STATUS*
NppNET::InitializeNetwork();
```

Called by the NppNET component to initialize the network.

Overview

Condition	Description
Pre-condition	None
Action	Calls user supplied global routine, NppInitializeNetwork(), that is responsible for creating the memory pool used by Nucleus NET, and calling the NU_Init_Net routine with a pointer to that memory pool.
Post-condition	The networking stack is ready for use.

Parameters

None

Return Value

Return Value	Overview
NU_SUCCESS	Indicates successful completion of the service.
NU_UNAVAILABLE	The member failed because a needed resource was unavailable.

Example

```
pNetworkComponent->InitializeNetwork();
```



NppNET : :NppNET

NppNET

NppNET : :NppNET () ;

Constructor

Overview

Condition	Description
Pre-condition	None
Action	None
Post-condition	The object exists.

Parameters

None

Return Value

None

Example

```
NppNET ( ) ;
```



NppNET::RegisterDevice

```
virtual
STATUS
NppNET::RegisterDevice
(
    NuNetDevice* pDevice
);
```

Registers an instance of an `NuNetDevice` derived object with the network stack.

Overview

Condition	Description
Pre-condition	None
Action	Adds the device driver to the network stack by calling the <code>NU_Init_Devices</code> routine.
Post-condition	The device driver has successfully been inserted into the network stack and is ready for use.

Parameters

Parameter	Overview
<code>pDevice</code>	A pointer to the device object to register.

Return Value

Return Value	Overview
<code>NU_SUCCESS</code>	Indicates successful completion of the service.
<code>NU_UNAVAILABLE</code>	The member failed because a needed resource was unavailable.

Example

```
MyATMDevice* pDevice = new MyATMDevice;
pNetworkComponent->RegisterDevice( pDevice );
```



NppNET::UnRegisterDevice

```
virtual
STATUS
NppNET::UnRegisterDevice
(
    NuNetDevice* pDevice
);
```

Nucleus NET does not currently support removal of devices that have been registered with the network stack. In future versions, this member will remove a given device.

Overview

Condition	Description
Pre-condition	None
Action	Shuts down a network device and removes it from the networking stack.
Post-condition	None, this routine is currently unsupported.

Parameters

Parameter	Overview
pDevice	A pointer to a derived network device.

Return Value

Return Value	Overview
NU_SUCCESS	Indicates successful completion of the service.
value <0	The operation failed and an error code was returned.

Example

```
MyEthernetDevice* pDevice;
...
pNetworkComponent->UnRegisterDevice( pDevice );
```



class NuNetDevice

Class that encapsulates the Nucleus NET device driver. This is meant to be a base class for user developed device drivers that take advantage of object-oriented techniques.

Public Member Functions

Member	Overview
~NuNetDevice	Destructor
GetDeviceStruct	Returns a const pointer to this devices NU_DEVICE structure.
GetDeviceStruct	Returns a pointer to this devices NU_DEVICE structure.
GetDriverOptions	Returns the driver options contained in the internal NU_DEVICE struct member.
GetFlags	Returns the flags contained in the internal NU_DEVICE struct member.
GetGateway	Constructs and returns an IPAddress object that contains the gateway IP address in the NU_DEVICE struct member.
GetIP	Constructs and returns an IPAddress object that contains the IP address in the NU_DEVICE struct member.
GetName	Returns the name pointer contained in the NU_DEVICE struct member.
GetSubnet	Constructs and returns an IPAddress object that contains the subnet IP address in the NU_DEVICE struct member.
NuNetDevice	Constructor
SetDriverOptions	Sets the driver options field inside of the internal NU_DEVICE.
SetFlags	Sets the flags to be supported for your device to the given value.
SetGateway	Sets the gateway IP address member of the NU_DEVICE structure.
SetInitRoutine	Called to set the C linkage initialization routine that Nucleus NET calls for a given device.
SetIP	Sets the IP address member of the NU_DEVICE structure.
SetName	Sets the name pointer inside of the device structure.
SetSubnet	Sets the subnet IP address member of the NU_DEVICE structure.



Protected Member Functions

Member	Overview
GetDeviceEntryStruct	Returns a pointer to the DV_DEVICE_ENTRY structure member pointer.
Init	Derived devices override this member to perform one-time initialization of the network device.
Input	Reads packets from the device and pass them to the appropriate upper layer
Ioctl	This function processes IOCTL requests from the upper layer protocol.
Output	The device output function puts a MAC layer header onto an outgoing packet and if necessary resolves the MAC layer address via ARP.
SetInputRoutine	Sets a pointer to the routine to use for reading available network buffers and passing them to higher layers.
SetOutputRoutine	Sets a pointer to the routine to use for doing MAC layer device.
Start	Called to transmit a packet.



NuNetDevice::~~NuNetDevice

```
virtual  
NuNetDevice::~~NuNetDevice();
```

Destructor

Overview

Condition	Description
Pre-condition	None
Action	None
Post-condition	The object doesn't exist.

Parameters

None

Return Value

None

Example

Destructors are called automatically when the object goes out of scope or is deleted.



NuNetDevice::GetDeviceEntryStruct

```
inline  
DV_DEVICE_ENTRY*  
NuNetDevice::GetDeviceEntryStruct();
```

Returns a pointer to the DV_DEVICE_ENTRY structure member pointer. This structure is allocated by Nucleus NET and the Nucleus C++ NET component assigns this member during driver registration.

Overview

Condition	Description
Pre-condition	RegisterDevice has been called. It is responsible for setting this pointer as the structure does not exist before Nucleus NET creates the structure.
Action	Returns the pDeviceEntryStruct member.
Post-condition	None

Parameters

None

Return Value

Return Value	Overview
AvalidPointer	A pointer to the internal data member is returned.

Example

```
STATUS  
MyDevice::Start()  
{  
    // This routine transmits a packet.  
    InterruptSystem::Disable();  
    outp( GetDeviceStruct()->dev_io_addr, m_data );  
}
```



NuNetDevice::GetDeviceStruct

```
inline
NU_DEVICE*
NuNetDevice::GetDeviceStruct();
```

Called to get access to an NuNetDevice objects NU_DEVICE member. This structure can then be manually modified.

Overview

Condition	Description
Pre-condition	None
Action	Returns the internal pointer.
Post-condition	None

Parameters

None

Return Value

Return Value	Overview
aValidPointer	A pointer to the internal data member is returned.

Example

```
NuEthernetDevice *pDevice = new NuEthernetDevice;
NU_DEVICE* pDevStruct = pDevice->GetDeviceStruct();
pDevStruct->uart.baud_rate = 115200;
```



NuNetDevice::GetDeviceStruct

```
inline  
NU_DEVICE*  
NuNetDevice::GetDeviceStruct()  
const;
```

Called to get access to an NuNetDevice objects NU_DEVICE member. The returned pointer cannot be modified.

Overview

Condition	Description
Pre-condition	None
Action	Returns the address of the internal deviceStruct member.
Post-condition	None

Parameters

None

Return Value

Return Value	Overview
AValidPointer	A const pointer to the internal data member is returned.

Example

```
const NuNetDevice* pDevice = GetTheRelevantDevice();  
  
INT32 par = pDevice->GetDeviceStruct()->uart.parity;
```



NuNetDevice::GetDriverOptions

```
inline  
UINT32  
NuNetDevice::GetDriverOptions()  
const;
```

Returns the driver options contained in the internal NU_DEVICE struct member.

Overview

Condition	Description
Pre-condition	None
Action	Returns deviceStruct.dv_driver_options.
Post-condition	None

Parameters

None

Return Value

Return Value	Overview
validData	The current value of the driver options field is returned.

Example

```
UINT32 options = mydevice.GetDriverOptions();
```



NuNetDevice::GetFlags

```
inline  
INT32  
NuNetDevice::GetFlags()  
const;
```

Returns the flags contained in the internal NU_DEVICE struct member.

Overview

Condition	Description
Pre-condition	None
Action	Returns deviceStruct.dv_flags.
Post-condition	None

Parameters

None

Return Value

Return Value	Overview
validData	The current value of the dv_flags member is returned.

Example

```
INT32 flags = mydevice.GetFlags();  
flags |= DV_MULTIPPOINT;  
mydevice.SetFlags();
```



NuNetDevice::GetGateway

```
inline
IPAddress
NuNetDevice::GetGateway()
const;
```

Constructs and returns an IPAddress object that contains the gateway IP address in the NU_DEVICE struct member. Currently the gateway field is not used by Nucleus NET. It should be noted that this routine creates IPAddress objects on the stack, which means that calling it in any code in which speed is important is a bad idea. A better approach would be to create an IPAddress object as a member in your device object, and initialize it in the Init routine with a call to GetIP. The other alternative is to use the deviceStruct.dv_gw field directly.

Overview

Condition	Description
Pre-condition	None
Action	Constructs an IPAddress object and initializes it to the value of deviceStruct.dv_gw.
Post-condition	None

Parameters

None

Return Value

Return Value	Overview
validObject	A new IPAddress object with the contained IP address is returned.

Example

```
IPAddress gwip = mydevice.GetGateway();
```



NuNetDevice::GetIP

```
inline
IPAddress
NuNetDevice::GetIP()
const;
```

Constructs and returns an `IPAddress` object that contains the IP address in the `NU_DEVICE` struct member. It should be noted that this routine creates `IPAddress` objects on the stack, which means that calling it in any code in which speed is important is a bad idea. A better approach would be to create an `IPAddress` object as a member in your device object, and initialize it in the `Init` routine with a call to `GetIP`. The other alternative is to use the `deviceStruct.dv_ip_addr` field directly.

Overview

Condition	Description
Pre-condition	None
Action	Constructs an <code>IPAddress</code> object and initializes it to the value of <code>deviceStruct.dv_ip_addr</code> .
Post-condition	None

Parameters

None

Return Value

Return Value	Overview
<code>validObject</code>	A new <code>IPAddress</code> object with the contained IP address is returned.

Example

```
IPAddress deviceip = mydevice.GetIP();
```



NuNetDevice::GetName

```
inline
CHAR*
NuNetDevice::GetName()
const;
```

Returns the name pointer contained in the internal `NU_DEVICE` struct member.

Overview

Condition	Description
Pre-condition	None
Action	Returns <code>deviceStruct.dv_name</code> .
Post-condition	None

Parameters

None

Return Value

Return Value	Overview
<code>aValidPointer</code>	A <code>const</code> pointer to the internal data member is returned.

Example

```
const CHAR* name = mydevice.GetName();
cout << "the device name " << name << endl;
```



NuNetDevice::GetSubnet

```
inline  
IPAddress  
NuNetDevice::GetSubnet()  
const;
```

Constructs and returns an `IPAddress` object that contains the subnet IP address in the `NU_DEVICE` struct member. It should be noted that this routine creates `IPAddress` objects on the stack, which means that calling it in any code in which speed is important is a bad idea. A better approach would be to create an `IPAddress` object as a member in your device object, and initialize it in the `Init` routine with a call to `GetIP`. The other alternative is to use the `deviceStruct.dv_subnet_mask` field directly.

Overview

Condition	Description
Pre-condition	None
Action	Constructs an <code>IPAddress</code> object and initializes it to the value of <code>deviceStruct.dv_subnet_mask</code> .
Post-condition	None

Parameters

None

Return Value

Return Value	Overview
<code>ValidObject</code>	A new <code>IPAddress</code> object with the contained IP address is returned.

Example

```
IPAddress subnet = mydevice.GetSubnet();
```



NuNetDevice::Init

```
virtual
STATUS
NuNetDevice::Init()=0;
```

Derived devices must override this member to perform one-time initialization of the network device. Responsible for Initializing all required function pointers and data fields in the DV_DEVICE_ENTRY structure that Nucleus NET uses to interface to us. Any one time device TYPE initialization should be done as well as any that is specific to this INSTANCE of the device. This includes registering any necessary LISR's or HISR's with the kernel. When this routine exits, the device should be ready to transmit and receive data. Return NU_SUCCESS if all goes well, a negative number if there is a problem.

Overview

Condition	Description
Pre-condition	Nucleus C++ NET has been initialized.
Action	Called by Nucleus NET when a device is registered to initialize it. It is assumed that when it returns, the device is ready to be used.
Post-condition	The device is ready to be used to transmit and receive data.

Parameters

None

Return Value

Return Value	Overview
NU_SUCCESS	Indicates successful completion of the service.
value < 0	The open operation failed for some reason. If you want to be able to return meaningful error codes to the thread that is initializing the device, return them as a negative number here.



Example

```
// In this example, we want to write our own
// ethernet driver.  To do this, we create the
// HISRS and LISR's that we need, and use the
// ethernet input and output routines that are
// supplied by Nucleus NET.
extern "C"
{
    STATUS NET_Ether_Input (VOID);

    STATUS NET_Ether_Send( NET_BUFFER* buf_ptr,
                           DV_DEVICE_ENTRY *device,
                           SCK_SOCKADDR_IP *dest,
                           RTAB_ROUTE *ro );
}

STATUS
MyEthernetDevice::Init()
{
    // use Nucleus NET MAC layer input and
    // output routines.

    SetInputRoutine( NET_Ether_Input );
    SetOutputRoutine( NET_Ether_Send );

    // Create interrupt objects we need.  Set
    // them up to work with this device instance.
    pMyTxHisr = new MyEthernetTxHisr( this );
    pMyRxHisr = new MyEthernetRxHisr( this );
    pLisr      = new MyEthernetLisr( this );

    // set our hardware up according to
    // Parameters passed in to our constructor
    ConfigureHardware();

    return( NU_SUCCESS );
}
```



NuNetDevice::Input

```
virtual
STATUS
NuNetDevice::Input();
```

Called by Nucleus NET in response to an event being set that indicates that there is received data available. Inputs responsibility is to demultiplex packets from the device and pass them to the appropriate upper layer protocol. It is assumed by Nucleus NET that one copy of Input is used for each different MAC layer. That means that all ethernet devices should share the same input routine. (In fact, for ethernet devices, this routine is provided by Nucleus NET, it is called `NET_Ether_Input`) In terms of how this translates to C++, Input should probably call a static member that is shared by all instances of a given device (if you had 5 ethernet devices, they would all share one output routine). If you want to call a static routine for input, add a static routine to your class, and in your `Init` routine, call `SetInputRoutine` with the address of your static input routine. The default implementation calls the virtual Input routine for optimum flexibility as far as the design goes.

Overview

Condition	Description
Pre-condition	There are data packets available. Most likely placed in the global buffers by interrupt service routines that are taking the packets from the wire.
Action	Default implementation does nothing.
Post-condition	Data has been removed from the input buffers decoded and sent up the stack.

Parameters

None

Return Value

Return Value	Overview
<code>NU_SUCCESS</code>	Indicates successful completion of the service.
<code>value < 0</code>	The operation failed and an error code was returned.



Example

```

// In this example, we parse incoming packets,
// removing our specialized framing and then pass
// them up to higher layers.
STATUS
MyFunkySerialDevice::Input()
{
    UINT16  packetType;
    UINT8*  pPacket;

    DV_DEVICE_ENTRY* device = GetDeviceEntryStruct();

    // Get the pointer to the packet
    pPacket = (UINT8*)( MEM_Buffer_List.head->data_ptr );

    // What type of packet is this?
    packetType = GET16( pPacket, FUNKY_SERIAL_TYPE_OFFSET );

    // Strip the framing from the packet.
    DecodeTheFunkyFraming( pPacket );

    // What to do with it?
    switch( packetType )
    {
        case EIP:
        {
            // This is an IP packet.
            IP_Interpret
            (
                (IPLAYER*)MEM_Buffer_List.head->data_ptr,
                device, MEM_Buffer_List.head
            );
        }
        break;

        default:
        {
            // Some unknown packet, drop it...
            MEM_Buffer_Chain_Free
            (
                &MEM_Buffer_List, &MEM_Buffer_Freelist
            );
        }
        break;
    }

    return( NU_SUCCESS );
}

```



NuNetDevice::Ioctl

```
virtual
STATUS
NuNetDevice::Ioctl
(
    INT option,
    DV_REQ* pDeviceRequest
)=0;
```

This function processes IOCTL requests from the upper layer protocol. Currently the only IOCTL requests used by Nucleus NET are the addition and deletion of multicast addresses to the list of multicast addresses that are to be received by the driver. (DEV_ADDMULTI and DEV_DELMULTI)

Overview

Condition	Description
Pre-condition	Nucleus C++ NET has been initialized. The device has been registered.
Action	Adds and removes device specific options as necessary.
Post-condition	The option has been changed as specified.

Parameters

Parameter	Overview
option	A field that identifies the particular option that is to be set for this device. Usually a #define for that particular device.
pDeviceRequest	A pointer to the device request structure which contains fields that house the data needed to set a given option.

Return Value

None



Example

```
STATUS
MyNetDevice::Ioctl( INT option, DV_REQ *d_req)
{
    STATUS          status = NU_SUCCESS;
    DV_DEVICE_ENTRY* dev = GetDeviceEntryStruct();

    switch (option)
    {
        case DEV_ADDMULTI :

            // Join the ethernet multicast group.
            status = NET_Add_Multi(dev, d_req);
            break;

        case DEV_DELMULTI :

            // Join the ethernet multicast group.
            status = NET_Del_Multi(dev, d_req);
            break;

        default :
            status = NU_INVALID;
            break;
    }
    return (status);
}
```



NuNetDevice::NuNetDevice

```
NuNetDevice::NuNetDevice();
```

Constructor. NuNetDevice is an abstract base class for device objects. As such it is never directly instantiated.

Overview

Condition	Description
Pre-condition	None
Action	Initializes the base class.
Post-condition	None

Parameters

None

Return Value

None

Example

This is an abstract class and is never directly created.



NuNetDevice::Output

```

virtual
STATUS
NuNetDevice::Output
(
    NET_BUFFER* pBuffer,
    SCK_SOCKADDR_IP* pDest,

    RTAB_ROUTE* pRoute
);

```

The device output function puts a MAC layer header onto an outgoing packet and if necessary resolves the MAC layer address via ARP. This function differs for MAC layers of different types but is the same for all MAC layers of the same type. Like `Input`, it is assumed that there is one copy of output shared by all devices that share a given MAC layer. The default implementation is to call this virtual `Output` routine which you can overload if desired, but you can also call `SetOutputRoutine` with a pointer to a static output function if so desired (so that there is not a V-table access and it is thus more efficient). Whatever the output function is, whether a static function or whatever, it typically does the MAC layer framing of the outgoing packet and then calls `Start`, to do the actual physical transmission of the data.

Overview

Condition	Description
Pre-condition	Nucleus C++ NET has been initialized. This device has been registered with the network.
Action	The device output function puts a MAC layer header onto an outgoing packet and if necessary resolves the MAC layer address via ARP.
Post-condition	Relevant data has been transmitted.

Parameters

Parameter	Overview
<code>pBuffer</code>	A pointer to the buffer to send.
<code>pDest</code>	Destination information.
<code>pRoute</code>	Pointer to the route to use for this device.

Return Value

Return Value	Overview
<code>NU_SUCCESS</code>	Indicates successful completion of the service.
<code>value < 0</code>	The operation failed and an error code was returned.



Example

```

// if you wanted to write an virtual overload of
// Output, it might look like this.
STATUS
MyFunkySerialDevice::Output( NET_BUFFER* pBuffer,
                             SCK_SOCKADDR_IP* pDest,
                             RTAB_ROUTE* pRoute)
{
    STATUS status;

    // check the route
    if (!RouteIsOk( pDest, pRoute ))
        return NU_HOST_UNREACHABLE;

    // reframe the packet with our funky serial
    // protocol framing. (HDLC, SDLC, wierd encryption,
    // whatever we want...
    FrameTheBuffer( pBuffer );

    // transmit the buffer using the virtual
    // Start routine for this device.
    status = this->Start( pBuffer );

    // The packet has been transmitted. Deallocate
    // the buffer.

```



NuNetDevice::SetDriverOptions

```
inline  
VOID  
NuNetDevice::SetDriverOptions  
(  
    UINT32 options  
);
```

Sets the driver options field inside of the internal `NU_DEVICE` structure. Driver options are completely driver specific, Nucleus NET makes no use of them at all, they are purely used for communicating options from the application to the driver.

Overview

Condition	Description
Pre-condition	None
Action	Sets <code>deviceStruct.dv_driver_options</code> .
Post-condition	None

Parameters

Parameter	Overview
<code>options</code>	A 32 bit number containing the desired options.

Return Value

None

Example

```
mydevice.SetDriverOptions( PUMP_DATA_REAL_FAST );
```



NuNetDevice::SetFlags

```
inline
VOID*
NuNetDevice::SetFlags
(
    INT32 flags
);
```

Sets the flags to be supported for your device to the given value. See Nucleus NET file `DEV.H` and Nucleus NET documentation for a list of valid flags for NET devices.

Overview

Condition	Description
Pre-condition	None
Action	Sets the <code>dv_flags</code> member of the internal <code>deviceStruct</code> member.
Post-condition	None

Parameters

Parameter	Overview
<code>flags</code>	The new <code>flags</code> value.

Return Value

None

Example

```
MyEthernetDevice::MyEthernetDevice()
{
    SetName("bob");
    SetFlags( DV_BROADCAST | DV_MULTICAST );
    SetIP( IPAddress(200,100,50,0) );
    SetSubnet( IPAddress(255,255,255,0) );

    SetEthernetIrq( irq );
    SetEthernetIOAddress( io );
    SetEthernetSharedMemoryAddress( 0 );// unused
}
```



NuNetDevice::SetGateway

```
inline  
VOID  
NuNetDevice::SetGateway  
(  
    const IPAddress& ip  
);
```

Sets the gateway IP address member of the NU_DEVICE structure. Note that this is not currently used by Nucleus.

Overview

Condition	Description
Pre-condition	None
Action	Sets the <code>deviceStruct.dv_gw</code> member.
Post-condition	None

Parameters

Parameter	Overview
<code>ip</code>	Reference to an IP address to be copied.

Return Value

None

Example

```
IPAddress gw(200,100,50,0);  
mydevice.SetGateway( gw );
```



NuNetDevice::SetInitRoutine

```
inline
VOID
NuNetDevice::SetInitRoutine
(
    STATUS (*dv_init) (DV_DEVICE_ENTRY *)
);
```

Called to set the C linkage initialization routine that Nucleus NET calls for a given device. By default, the `NuNetDeviceConstructor` sets this pointer up so that your derived devices Init routine gets called instead of a global routine. This routine is called in the derived class `WrapperDevice` which is designed to be a no overhead C++ wrapper for a pre-developed Nucleus NET C device driver.

Overview

Condition	Description
Pre-condition	None
Action	Sets the <code>deviceStruct.dv_init</code> member.
Post-condition	None

Parameters

Parameter	Overview
<code>(*dv_init)</code> <code>(DV_DEVICE_ENTRY *)</code>	A pointer to a C linkage initialization function for the device.

Return Value

None



Example

```
// This is an example of the Constructor for the class
// VnetDevice which is derived from WrapperDevice, a
// class derived from NuNetDevice that is meant for
// wrapping an existing Nucleus C network driver.

extern "C"
{
    // This is the device specific "C" low level
    // driver initialization routine for Nucleus VNET

    extern STATUS VDRV_Init(DV_DEVICE_ENTRY *device);
}

VnetDevice::VnetDevice( const CHAR* pDevName, const IPAddress& ip )
{
    SetName(pDevName);
    SetInitRoutine( VDRV_Init );
    SetIP( ip );
    SetSubnet( IPAddress(200,100,50,0) );

    SetFlags( 0 );                // unused
    SetEthernetIrq( 0 );          // unused
    SetEthernetIOAddress( 0 );    // unused
    SetEthernetSharedMemoryAddress( 0 ); // unused
}
```



NuNetDevice::SetInputRoutine

```
inline
VOID
NuNetDevice::SetInputRoutine();
```

Sets a pointer to the routine to use for reading available network buffers and passing them to higher layers. By default this pointer is to the `NuNetDevice` static routine `NuNetDevice::Input` which will call your derived `NuNetDevice` objects virtual `Input` routine. You can set it to whatever you wish (In your `init` routine) if you don't want to have a v-table lookup here. See also `NuNetDevice::Input`

Overview

Condition	Description
Pre-condition	You are calling this from inside of your <code>init</code> routine. Before this point, the member that you are setting doesn't exist because it is created by Nucleus NET.
Action	Sets the <code>pDeviceEntryStruct->dev_input</code> member.
Post-condition	The pointer to the function to call has been set.

Parameters

None

Return Value

None



Example

```
// In this example, we want to write our own
// ethernet driver. To do this, we create the
// HISRS and LISR's that we need, and use the
// ethernet input and output routines that are
// supplied by Nucleus NET.
extern "C"
{
    STATUS NET_Ether_Input (VOID);

    STATUS NET_Ether_Send(NET_BUFFER* buf_ptr,
                          DV_DEVICE_ENTRY *device,
                          SCK_SOCKETADDR_IP *dest,
                          RTAB_ROUTE *ro);
}

STATUS
MyEthernetDevice::Init()
{
    // use Nucleus NET ethernet routines.
    SetInputRoutine( NET_Ether_Input );
    SetOutputRoutine( NET_Ether_Send );

    // Create interrupt objects we need. Set
    // them up to work with this device instance.
    pMyTxHsr = new MyEthernetTxHsr( this );
    pMyRxHsr = new MyEthernetRxHsr( this );
    pLisr     = new MyEthernetLisr( this );

    ConfigureHardware();

    return( NU_SUCCESS );
}
```



NuNetDevice::SetIP

```
inline
VOID
NuNetDevice::SetIP
(
const IPAddress& ip
);
```

Sets the IP address member of the `NU_DEVICE` structure. Note that it is assumed that this happens BEFORE registration finishes for the device. (Before the `Init` routine is completed). The IP address cannot be changed once the device is registered.

Overview

Condition	Description
Pre-condition	None
Action	Sets the <code>deviceStruct.dv_ip_addr</code> member.
Post-condition	None

Parameters

Parameter	Overview
<code>ip</code>	A reference to the IP address to copy.

Return Value

None

Example

```
IPAddress ip(200,100,50,0);
mydevice.SetIP( ip );
```



NuNetDevice::SetName

```
inline
VOID
NuNetDevice::SetName
(
    const CHAR* pName
);
```

Sets the name pointer inside of the device structure.

Overview

Condition	Description
Pre-condition	None
Action	Sets the dv_name member inside of the NU_DEVICE structure.
Post-condition	None

Parameters

Parameter	Overview
pName	A pointer to the name for the device.

Return Value

None

Example

```
NuEthernetDevice * pMyDevice = new NuEthernetDevice;
pMyDevice->SetName("bob");
```



NuNetDevice::SetOutputRoutine

```

inline
VOID
NuNetDevice::SetOutputRoutine
(
    STATUS (__cdecl *dev_output)( args )
);

```

Sets a pointer to the routine to use for doing MAC layer device output. By default this pointer is to `NuNetDevice::output` which will call your derived `NuNetDevice` objects virtual Output routine. In the case of ethernet devices, Nucleus NET provides a common routine to do ethernet packet framing. This shared output routine frames the ethernet packet and then calls your derived devices. Start routine to do the actual transmission of data on the wire.

Overview

Condition	Description
Pre-condition	You are calling this routine from within your derived devices <code>init</code> routine.
Action	Sets the <code>pDeviceEntryStruct->dev_output</code> routine pointer.
Post-condition	The device output routine pointer has been set.

Parameters

Parameter	Overview
<code>(__cdecl*dev_output)</code> <code>(args)</code>	A pointer to the output function. The function must have this prototype <code>:STATUS</code>

Return Value

None



Example

```
// In this example, we want to write our own
// ethernet driver. To do this, we create the
// HISRS and LISR's that we need, and use the
// ethernet input and output routines that are
// supplied by Nucleus NET.
extern "C"
{
    STATUS NET_Ether_Input (VOID);

    STATUS NET_Ether_Send(NET_BUFFER* buf_ptr,
                          DV_DEVICE_ENTRY *device,
                          SCK_SOCKADDR_IP *dest,
                          RTAB_ROUTE *ro);
}
STATUS
MyEthernetDevice::Init()
{
    // use Nucleus NET ethernet routines.
    SetInputRoutine( NET_Ether_Input );
    SetOutputRoutine( NET_Ether_Send );

    // Create interrupt objects we need. Set
    // them up to work with this device instance.
    pMyTxHsr = new MyEthernetTxHsr( this );
    pMyRxHsr = new MyEthernetRxHsr( this );
    pLisr     = new MyEthernetLisr( this );

    ConfigureHardware();

    return( NU_SUCCESS );
}
```



NuNetDevice::SetSubnet

```
inline
VOID
NuNetDevice::SetSubnet
(
const IPAddress& ip
);
```

Sets the subnet IP address member of the NU_DEVICE structure. Note that it is assumed that this happens BEFORE registration finishes for the device. (Before the Init routine is completed). The subnet cannot be changed once the device is registered.

Overview

Condition	Description
Pre-condition	None
Action	Sets the <code>deviceStruct.dv_subnet_mask</code> member.
Post-condition	None

Parameters

Parameter	Overview
<code>ip</code>	An IP address to be copied for the subnet field.

Return Value

None

Example

```
IPAddress subnet(255,255,255,0);
mydevice.SetSubnet( subnet );
```



NuNetDevice::Start

```
virtual
STATUS
NuNetDevice::Start
(
    NET_BUFFER* pBuffer
)=0;
```

The purpose of the device start function is to transmit a packet. The start function is provided a complete packet by the upper layer software, and its responsibility is to place the packet onto the physical medium. The parameter is a pointer to the buffer chain containing the packet to be sent. NU_SUCCESS should be returned upon success. A negative value should be returned otherwise.

Overview

Condition	Description
Pre-condition	The packet has been formatted by the upper protocol layers and is ready for the wire.
Action	Called by the output routine to transmit the packet.
Post-condition:	The packet has been transmitted (or is queued to be transmitted if you are using a communications co-processor or interrupts or whatever to do the actual work of transmitting).

Parameters

Parameter	Overview
pBuffer	The network buffer to send.

Return Value

None

Example

```
STATUS MyEtherDevice::Start( NET_BUFFER* data )
{
    return CopyDataIntoHardware( data );
}
```



class NuRawSocket : public NuSocket

A derived socket that works directly with the Network layer, bypassing the Transport facilities that TCP and UDP use.

Public Member Functions

Member	Overview
~NuRawSocket	Destructor
Create	Allocates space in the network stack for this socket.
NuRawSocket	Default Constructor
RecvFromRaw	Receives data from the network during a raw IP transfer.
RecvFromRaw	Receives data from the network during a raw IP transfer.
SendToRaw	Sends data via raw IP.
SendToRaw	Send via Raw IP.
SetAppLayerDoesIPHeader	Controls IP header completion.



NuRawSocket : ~NuRawSocket

```
virtual  
VOID*  
NuRawSocket : ~NuRawSocket () ;
```

Destructor

Overview

Condition	Description
Pre-condition	None
Action	None
Post-condition	The object does not exist.

Parameters

None

Return Value

None

Example

Destructors are called automatically when the object goes out of scope or is deleted.



NuRawSocket::Create

```
virtual
BOOL
NuRawSocket::Create();
```

Create is used for explicit creation of a socket after the Constructor has been called. It is used to allocate space in the network stack for this instance of NuSocket derived object. Creation of a socket is then always a two part sequence, First constructing the socket, then calling Create to initialize it. Create can be called again for the socket AFTER a call to Close() or Abort(). In this way you in essence "re-use" the socket object without having to allocate and deallocate.

Overview

Condition	Description
Pre-condition	The object has been constructed.
Action	Allocates a new socket descriptor from Nucleus NET for this instance.
Post-condition	Space in the network stack has been allocated for this socket instance. The socketd member contains the new socket descriptor.

Parameters

None

Return Value

Return Value	Overview
FALSE	If not successful.
TRUE	If successful.



Example

```
NuRawSocket workersocket;  
SocketAddress dest;  
  
// www.mysite.com  
dest.SetIP( 200,100,50,0 );  
dest.SetPort( 0001 );  
  
while( 1 )  
{  
    // allocate space in the network stack  
    workersocket.Create();  
  
    workersocket.SendToRaw(dataptr,datasize,dest);  
  
    // deallocate the socket from the network stack  
    workersocket.Close();  
  
    // sleep for a second and do it again  
    Task::Sleep( systemClockFrequency );  
}
```



NuRawSocket::NuRawSocket

NuRawSocket

NuRawSocket::NuRawSocket(INT16 protocol);

Constructor. Constructs a Raw IP specific NuSocket derived object. Please see NuRawSocket::Create for more information.

Overview

Condition	Description
Pre-condition	None
Action	None
Post-condition	The object exists.

Parameters

Parameter	Overview
protocol	The raw protocol to use. Allowable values are: IPPROTO_HELLO HELLO routing protocol IPPROTO_OSPF OSPF routing protocol IPPROTO_RAW Raw IP protocol 0 Raw IP wildcard protocol

Return Value

None

Example

```
NuRawSocket* pSocket = new NuRawSocket(IPPROTO_RAW);

// allocate space in the TCP/IP stack for this
// socket.
pSocket->Create();

if ( pSocket->IsValidSocket() )
{
    INT32 sent;
    sent = pSocket->SendToRaw( pData, 100, address );
    ...
}
else
{
    // the allocation of the socket in
    //the networking stack failed.
```



NuRawSocket::RecvFromRaw

```
inline
INT32
NuRawSocket::RecvFromRaw
(
    CHAR* pData,
    UINT16 nbytes,
    SocketAddress* pFrom = NULL
);
```

Receive data from the network during a raw IP transfer. Raw IP works directly with the network layer of the stack.

Overview

Condition	Description
Pre-condition	Nucleus C++ NET has been initialized. This object has been constructed and <code>Create</code> has been called.
Action	<code>NU_Recv_From_Raw</code> service is called. The underlying service can suspend the calling thread until data is available to be read.
Post-condition	If successful, data has been copied into the data buffer and the number of bytes received is returned. If unsuccessful, an error code has been returned.

Parameters

Parameter	Overview
<code>pData</code>	Contains a pointer to the data buffer to place the received data in.
<code>nbytes</code>	contains the maximum number of bytes that can be written into the buffer.
<code>pFrom</code>	A valid pointer to a <code>SocketAddress</code> object to fill out with the IP address and port of the sending socket.

Return Value

Return Value	Overview
<code>NU_INVALID_SOCKET</code>	This socket did not have a valid socket descriptor. (either it was corrupted or <code>Create</code> wasn't called).
<code>NU_NO_DATA_TRANSFER</code>	The data transfer was not completed.
<code>value > 0</code>	The actual number of bytes received is returned.



Example

```
NuRawSocket workersocket;
SocketAddress address;
INT32 bytesRecived;

// www.mysite.com
address.SetIP( 200, 100, 50, 0 );
address.SetPort( 80 );

while( 1 )
{
    // allocate space in the network stack
    workersocket.Create();

    bytesReceived = workersocket.RecvFromRaw( dataptr,
                                              datasize,&address );

    // deallocate the socket from the network stack
    workersocket.Close();

    // sleep for a second and do it again
    Task::Sleep( systemClockFrequency );
}
```



NuRawSocket::RecvFromRaw

```

inline
INT32
NuRawSocket::RecvFromRaw
(
    CHAR* pData,
    UINT16 nbytes,
    SocketAddress& from
)

```

Receives data from the network during a raw IP transfer. This overload requires a reference to a `SocketAddress` to receive from.

Overview

Condition	Description
Pre-condition	Nucleus C++ NET has been initialized.
Action	NU_Recv_From_Raw service is called. The underlying service can suspend the calling thread until data is available to be read.
Post-condition	If successful, data has been copied into the data buffer and the number of bytes received is returned. If unsuccessful, an error code has been returned

Parameters

Parameter	Overview
pData	A pointer to the data buffer to place the received data in.
nbytes	The maximum number of bytes that can be received.
from	A reference to the <code>SocketAddress</code> object containing the IP address and port to receive from.

Return Value

Return Value	Overview
NU_INVALID_SOCKET	This socket did not have a valid socket descriptor. (either it was corrupted or <code>Create</code> wasn't called).
NU_NO_DATA_TRANSFER	The data transfer was not completed.
value > 0	The actual number of bytes received is returned.

Example

```

NuRawSocket sock;
SocketAddress peeraddress;
CHAR sensordata[50];
...
sock.SetBlocking( TRUE ); // suspend until data

if ( sock.RecvFrom( sensordata, 10, peeraddress ) > 0 )
{

```



NuRawSocket::SendToRaw

```
inline
INT32
NuRawSocket::SendToRaw
(
    CHAR* pData,
    UINT16 nbytes,
    SocketAddress& dest
)
```

Sends data via the network layer directly, bypassing the transport layer (TCP or UDP).

Overview

Condition	Description
Pre-condition	Nucleus NET has been initialized, the <code>Create()</code> member has been called on this object.
Action	Calls the <code>NU_Send_To_Raw</code> service.
Post-condition	If successful, data has been copied from the data buffer into the network stack and the number of bytes written is returned. If unsuccessful, an error code has been returned.

Parameters

Parameter	Overview
<code>pData</code>	A pointer to the data to send.
<code>nbytes</code>	The number of bytes to send.
<code>dest</code>	A reference to a <code>SocketAddress</code> object containing the IP address and port to send to.

Return Value

Return Value	Overview
<code>NU_INVALID_ADDRESS</code>	The address passed in was most likely incomplete (i.e., missing the IP number).
<code>NU_INVALID_SOCKET</code>	This socket did not have a valid socket descriptor. (Either it was corrupted or <code>Create</code> wasn't called.)
<code>NU_NO_DATA_TRANSFER</code>	The data transfer was not completed.
<code>value > 0</code>	The actual number of bytes that were transferred is returned.



Example

```
CHAR mydata[100];  
NuRawSocket    socket;  
SocketAddress   dest;  
dest.SetName( "MYUDPCCLIENT" );  
dest.SetPort( nMyTaskId );  
dest.SetIP(200, 100, 50, 0 );  
...  
socket.SendToRaw( mydata, 100, dest );
```



NuRawSocket::SendToRaw

```
inline
INT32
NuRawSocket::SendToRaw
(
    CHAR* pData,
    UINT16 nbytes,
    SocketAddress* pDest
)
```

Sends data via raw IP, which uses the network layer directly, bypassing the transport layer (TCP or UDP)

Overview

Condition	Description
Pre-condition	Nucleus NET has been initialized, the <code>Create()</code> member has been called on this object.
Action	Calls the <code>NU_Send_To_Raw</code> service.
Post-condition	If successful, data has been copied from the data buffer into the network stack and the number of bytes written is returned. If unsuccessful, an error code has been returned.

Parameters

Parameter	Overview
<code>pData</code>	contains a pointer to the data buffer containing the data to send.
<code>nbytes</code>	contains the maximum number of bytes that can be written into the buffer.
<code>pDest</code>	A valid pointer to a <code>SocketAddress</code> object containing the IP address and port to send to.

Return Value

Return Value	Overview
<code>NU_INVALID_ADDRESS</code>	The address passed in was most likely incomplete(i.e., missing the IP number).
<code>NU_INVALID_SOCKET</code>	This socket did not have a valid socket descriptor. (either it was corrupted, or <code>Create</code> wasn't called.)
<code>NU_NO_DATA_TRANSFER</code>	The data transfer was not completed.
<code>value > 0</code>	The actual number of bytes that were transferred is returned.



Example

```
CHAR mydata[100];
SocketAddress  thataddress;
NuRawSocket    socket;

socket.Create();
thataddress.SetPort( 10 );
thataddress.SetIP(200, 100, 50, 0 );
...
socket.SendToRaw( mydata, 100, thataddress );
```



NuRawSocket::SetAppLayerDoesIPHeader

```
inline
STATUS
NuRawSocket::SetAppLayerDoesIPHeader
(
    BOOL bOn
);
```

Member that allows control over who is responsible for filling in the IP header for outgoing data, the stack or the user application.

Overview

Condition	Description
Pre-condition	Nucleus C++ NET has been initialized. The NuRawSocket object has been constructed and Create has been called.
Action	Calls the Setsockopt member.
Post-condition	Nucleus NET changes how it treats outgoing data depending on the parameter.

Parameters

Parameter	Overview
bOn	If TRUE, the stack assumes that the user application is filling in the IP header and that it is part of the data buffer in a call to SendToRaw. FALSE tells the stack to fill in the IP header automatically. Default value on raw socket creation is FALSE.

Return Value

Return Value	Overview
NU_INVALID_SOCKET	This socket did not have a valid socket descriptor. (either it was corrupted or Create wasn't called)
NU_SUCCESS	Indicates successful completion of the service.



Example

```
SocketAddress address;
char buffer[1024];
NuRawSocket rawsocket;
...
// allocate a new socket descriptor
rawsocket.Create();

// specify that we want to fill in the ip header
rawsocket.SetAppLayerDoesIPHeader( TRUE );

while( !done )
{
    // fill out the header and data and such here.
    Your_Routine_To_Build_The_Header( address, buffer
    rawsocket.SendToRaw(buffer,address,address);
    ...
}
// return the socket descriptor to the stack.
rawsocket.Close();
```



class NuSocket

The base socket class. This class is not created and used directly, but rather the derived classes `NuTcpSocket`, `NuUdpSocket` and `NuRawSocket` are used as they implement the sending and receiving functionality of the different protocols. This base class provides all shared functionality.

Public Member Functions

Member	Overview
<code>~NuSocket</code>	Destructor
<code>Abort</code>	Unconditionally and immediately shuts down the connection.
<code>Bind</code>	Binds a socket address to this socket.
<code>Close</code>	Closes the socket connection.
<code>Create</code>	Called to allocate a new socket instance in the network stack for this <code>NuSocket</code> derived object.
<code>CreateFromSocketDescriptor</code>	Called to map this instance of an object to a pre-existing socket descriptor. Thus wrapping an object around the normal socket API.
<code>EnableBroadcasting</code>	Called to enable broadcasting on this <code>NuSocket</code> derived socket.
<code>Fcntl</code>	Provides access to Nucleus blocking flags.
<code>FD_Check</code>	Checks to see if the bit corresponding to this socket in a <code>FD_Set</code> structure is set.
<code>FD_Reset</code>	Resets the bit in a <code>FD_Set</code> structure corresponding to this socket.
<code>FD_Set</code>	Sets the bit in a <code>FD_Set</code> structure corresponding to this socket.
<code>GetMulticastTTL</code>	Returns Time To Live for multicast packets sent on this socket.
<code>GetPeerAddress</code>	Returns our peers <code>IPAddress</code> and port number.
<code>GetPeerAddress</code>	Called to get our peer's IP address and port number from the network.
<code>GetSocketDescriptor</code>	Returns the Nucleus socket descriptor for this instance.
<code>Getsockopt</code>	Returns status of an option for a particular socket.
<code>IsDataAvailable</code>	Returns TRUE if there is data currently available.
<code>IsValid</code>	Checks internal socket descriptor for validity.
<code>NuSocket</code>	Constructor
<code>SetBlocking</code>	Set the suspension mode for the socket.
<code>Setsockopt</code>	Sets an option for this socket.
<code>WaitForData</code>	Suspends calling thread until data is available or a client is connecting.



Protected Member Functions

Member	Overview
CreateBase	Explicit socket creation.
NuSocket	Copy Constructor.
operator=	Assignment operator.
SetSocketDesc	Sets the internal socket descriptor member.

Static Public Class Member Functions

Member	Overview
FD_Init	Called to initialize an FD_SET structure.
GetSocketFor	Returns a pointer to an NuSocket object given a socket descriptor.
Initialize	One time initialization of the Static portions of this object.
Ping	Send a ping request to an IP address.



NuSocket::~NuSocket

```
virtual  
VOID*  
NuSocket::~NuSocket () ;
```

Destructor

Overview

Condition	Description
Pre-condition	None
Action	Checks to see if our connection has been closed. If it hasn't then call <code>Abort ()</code> to close the connection without suspension since we don't know where we have been called from.
Post-condition	The object does not exist.

Parameters

None

Return Value

None

Example

Destructors are called automatically when the object goes out of scope or is deleted.



NuSocket::Abort

```
virtual
STATUS
NuSocket::Abort();
```

This function immediately and unconditionally aborts a TCP or UDP connection. The functionality is the same as `Close`, except that `Close` more elegantly suspends the caller until the remote connection closes as well in the case of TCP.

Overview

Condition	Description
Pre-condition	None
Action	Removes this <code>NuSocket</code> object from the static array of created socket objects. Calls the <code>NU_Abort</code> service routine which sends a <code>RESET</code> to the remote host in the case of TCP, and frees all resources associated with this socket instance.
Post-condition	All resources associated with this socket object are released. This instance of the object is then unusable unless either <code>Create</code> or <code>CreateFromSocketDescriptor</code> is called to acquire another socket descriptor.

Parameters

None

Return Value

Return Value	Overview
<code>NU_INVALID_SOCKET</code>	This socket did not have a valid socket descriptor. (either it was corrupted or <code>Create</code> wasn't called)
<code>NU_NO_PORT_NUMBER</code>	No local port number was stored in the socket descriptor.
<code>NU_SUCCESS</code>	Indicates successful completion of the service.

Example

```
if ( critical_error_in_your_app() )
{
    // immediately end communications
    pSocket->Abort();
    delete pSocket;
    pSocket = NULL;
}
```



NuSocket::Bind

```

inline
SOCKETD_TYPE
NuSocket::Bind
(
    const SocketAddress* myaddress = NULL
);

```

Called to assign a local address to this socket. This is an overload that requires a pointer. It must be called by a server, whether a connection-oriented or connectionless transfer is being established. It only needs to be called by a client in the case of a connectionless transfer.

Overview

Condition	Description
Pre-condition	Nucleus NET has been initialized. The socket object exists and Create has been called.
Action	Calls the Bind(SocketAddress&) overload.
Post-condition	If successful, the socket descriptor for this instance is returned.
	If unsuccessful, a status code is returned.

Parameters

Parameter	Overview
myaddress	A valid pointer to the SocketAddress info to be bound to this socket.

Return Value

Return Value	Overview
NU_INVALID_SOCKET	This socket did not have a valid socket descriptor. (either it was corrupted or Create wasn't called).
value >= 0	The function succeeded, and the value contains the socket descriptor.

Example

```

SocketAddress servaddr;
servaddr.SetIP( 200, 100, 50, 1 );
servaddr.SetName("MYTCPSERVER");
servaddr.SetPort( 1002 );

if (serverSocket->Bind( &servaddr ) >= 0)
{
    ...
}

```



NuSocket::Close

```
virtual
STATUS
NuSocket::Close();
```

Closes the connection in the case of TCP, and deallocates the network resources for this socket in the case of UDP and RawIP.

Overview

Condition	Description
Pre-condition	Nucleus NET has been initialized, and <code>Create</code> has been called on this socket. <code>function</code> will suspend the calling thread until the socket connection is completely closed. In the case of connectionless, (UDP), the function closes the connection and returns immediately. If suspension is not acceptable, you can call <code>Abort</code> .
Post-condition	The socket has been closed. This instance of the object is then unusable unless either <code>CreateFromSocketDescriptor</code> or <code>Create</code> is called to acquire another socket descriptor.

This object instance is also removed from the static list of `NuSocket` objects.

Parameters

None

Return Value

Return Value	Overview
<code>NU_INVALID_SOCKET</code>	This socket did not have a valid socket descriptor. (Either it was corrupted or <code>Create</code> wasn't called.)
<code>NU_NO_PORT_NUMBER</code>	No local port number was stored in the socket descriptor.
<code>NU_SUCCESS</code>	Indicates successful completion of the service.

Example

```
NuTcpSocket* pSocket = new NuTcpSocket;
pSocket->Create();
if ( pSocket->Connect( serverAddress ) )
{
    pSocket->Send(data,length);
    pSocket->Close();
}
```



NuSocket::Create

```
virtual
BOOL
NuSocket::Create()=0;
```

Called to allocate a new socket instance in the network stack for an `NuSocket` derived object. `Create` is overridden by derived classes to explicitly create a socket after the Constructor has been called. See documentation for `NuTcpSocket::Create`, `NuUdpSocket::Create`, and `NuRawSocket::Create`.

Overview

Condition	Description
Pre-condition	Nucleus NET has been initialized, the derived socket object has been constructed.
Action	Derived classes allocate and setup a new socket instance in the network stack by overloading <code>Create</code> . <code>NuSocket</code> provides the protected routine <code>CreateBase</code> for derived objects to call to do this.
Post-condition	The derived socket object is ready to be used.

Parameters

None

Return Value

Return Value	Overview
FALSE	If not successful.
TRUE	If successful.

Example

```
// pure virtual and thus can't be called.
// See examples for NuTcpSocket::Create,
// NuUdpSocket::Create, and NuRawSocket::Create.
```



NuSocket::CreateBase

```
BOOL  
NuSocket::CreateBase  
(  
    INT16 type  
);
```

Explicit socket creation. Should only be called by derived class. Use derived class `Create` member.

Overview

Condition	Description
Pre-condition	Nucleus NET has been initialized.
Action	Allocates a new socket descriptor from Nucleus NET for this instance.
Post-condition	A socket descriptor for the new socket has been allocated within the network stack.

Parameters

Parameter	Overview
type	A 16 bit integer containing one of three predefined types of sockets to create. <code>NU_TYPE_STREAM</code> for TCP connections, <code>NU_TYPE_DGRAM</code> for UDP, and <code>NU_TYPE_RAW</code> for raw IP.

Return Value

Return Value	Overview
FALSE	If not successful.
TRUE	If successful.

Example

```
BOOL  
TcpSocket::Create()  
{  
    return( NuSocket::CreateBase( NU_TYPE_STREAM ) );  
}
```



NuSocket::CreateFromSocketDescriptor

```
virtual
BOOL
NuSocket::CreateFromSocketDescriptor
(
    SOCKETD_TYPE nSocketDesc
);
```

This version of `create` basically creates an object wrapper around a pre-existing socket descriptor. This would enable you to use `NuSocket` derived objects in an application that gives you a socket descriptor and not an `NuSocket` object, as in upgrading a legacy Nucleus C application to C++, or in working with Nucleus WebServ.

Overview

Condition	Description
Pre-condition	Nucleus C++ NET has been initialized.
Action	Calls the <code>GetPeerAddress(SocketAddress& address)</code> member.
Post-condition	The socket object is ready to be used. It should be noted that you must know the type and state of the underlying socket that you are wrapping with an object. Otherwise, it is possible to create a situation where the object doesn't work at all; (i.e. an <code>NuTcpSocket</code> instance that is really a <code>udp</code> socket underneath, in which case every function call would fail, or calling <code>Create</code> on a socket that has already been created via a call to <code>NU_Socket</code> directly).

Parameters

Parameter	Overview
<code>nSocketDesc</code>	Contains a valid pre-existing socket descriptor.

Return Value

Return Value	Overview
<code>FALSE</code>	If not successful.
<code>TRUE</code>	If successful.



Example

```
/* C code you are interfacing with does this...*/
INT16 socket_desc;
    socket_desc = NU_Socket( NU_FAMILY_IP, NU_TYPE_STREAM, NU_NONE );

...
// your C++ code
TcpSocket* mysocket = new TcpSocket;
mysocket->CreateFromSocketDescriptor( socket_desc );
// mysocket is ready to use...
```



NuSocket::EnableBroadcasting

```
inline
BOOL
NuSocket::EnableBroadcasting
(
    BOOL bEnable
);
```

Called to enable broadcasting on this `NuSocket` derived socket. Broadcasting is enabled by default.

Overview

Condition	Description
Pre-condition	Nucleus C++ NET has been initialized, the socket object has been constructed and <code>Create</code> has been called.
Action	Calls the <code>Setsockopt</code> service.
Post-condition	If successful, <code>Broadcasting</code> has been changed for this socket.

Parameters

Parameter	Overview
<code>bEnable</code>	If <code>TRUE</code> , broadcasting is enabled on this socket. If <code>FALSE</code> , broadcasting is disabled.

Return Value

Return Value	Overview
<code>FALSE</code>	If not successful.
<code>TRUE</code>	If successful.

Example

```
NuUdpSocket sock;
sock.Create();
// turn broadcasting off for this socket.
sock.EnableBroadcasting( FALSE );
```



NuSocket::Fcntl

```
inline
STATUS
NuSocket::Fcntl
(
    INT16 command,
    INT16 argument
);
```

This member exists to provide compatibility with future Nucleus NET options. Currently, it is only used to turn blocking on `Recv`, `Accept`, and `RecvFrom` on or off, so it is advised to use the `SetBlocking` member instead.

Overview

Condition	Description
Pre-condition	Nucleus C++ NET has been initialized. The <code>NuSocket</code> derived object has been constructed and <code>Create</code> has been called.
Action	Sets a given feature for this <code>NuSocket</code> derived object in the network stack.
Post-condition	If successful, the given flag has been set.

Parameters

Parameter	Overview
command	Currently the only valid option is <code>NU_SETFLAG</code> .
argument	Currently the only valid values are <code>NU_BLOCK</code> to enable blocking and <code>NU_FALSE</code> to disable blocking.

Return Value

Return Value	Overview
<code>NU_INVALID_SOCKET</code>	This socket did not have a valid socket descriptor. (either it was corrupted or <code>Create</code> wasn't called).
<code>NU_NO_ACTION</code>	No action was processed by this function.
<code>NU_SUCCESS</code>	Indicates successful completion of the service.

Example

```
// enable blocking on recv
pSocket->Fcntl( NU_SETFLAG, NU_BLOCK );

// disable blocking
pSocket->Fcntl( NU_SETFLAG, NU_FALSE );
```



NuSocket::FD_Check

```

inline
BOOL
NuSocket::FD_Check
(
    FD_SET* pSet
)
const;

```

Used in conjunction with the Nucleus C API routine `NU_Select` to check if data is available for this socket object. This functionality is used when more than one socket needs to be monitored for data simultaneously. If you only want to monitor data for one socket, see `WaitForData` or `IsDataAvailable`. See also `FD_Set`, and `FD_Reset`.

Overview

Condition	Description
Pre-condition	The object exists.
Action	Calls the <code>NU_FD_Check</code> service. Returns <code>TRUE</code> if the bit corresponding to this socket is set in the given set, <code>FALSE</code> otherwise.
Post-condition	None

Parameters

Parameter	Overview
<code>pSet</code>	A pointer to a Nucleus NET socket bitmap structure. This structure has a bit for every socket in the system.

Return Value

Return Value	Overview
<code>FALSE</code>	The bit in the <code>FD_SET</code> structure corresponding to our socket is NOT set.
<code>TRUE</code>	The bit in the <code>FD_SET</code> structure corresponding to our socket is set.



Example

```
STATUS          status;
FD_SET          set; // bitmap for sockets.
NuUdpSocket     socket1;
NuUdpSocket     socket2;

// assume that Create and Bind are called for
// both sockets.
...

// clear all bits in the socket bitmap
NuSocket::FD_Init(&set);

// set the bits corresponding to the two sockets
// that we want to watch for data on.
socket1.FD_Set( &set );
socket2.FD_Set( &set );

// suspend up to 1000 clock ticks waiting for data on
// the two sockets.
status = NU_Select(NSOCKETS,&set,NULL,NULL,1000);

if (status == NU_SUCCESS)
{
    // there was data received on at least one of the
    // sockets. Check which one it was

    if (socket1.FD_Check( &set ) )
    {
        // yes, there is data on socket1
        process_the_socket( socket1 );
    }
}
```



NuSocket::FD_Init

```
static
inline
VOID
NuSocket::FD_Init
(
    FD_SET* pSet
);
```

Called to initialize an `FD_SET` structure. Please see the documentation for `FD_Check` for a more thorough explanation of what this is and how you use it.

Overview

Condition	Description
Pre-condition	None
Action	Calls the <code>NU_FD_Init</code> service. Which clears all bits in the structure.
Post-condition	None

Parameters

Parameter	Overview
<code>pSet</code>	A pointer to an <code>FD_SET</code> structure.

Return Value

None

Example

```
// reset the bit that corresponds to us
    socket2.FD_Reset( &set );// Please see the example for
FD_Check for
    }// a more detailed example of use.
}
FD_SET new_data_set;
NuSocket::FD_Init( &new_data_set );
```



NuSocket::FD_Reset

```
inline  
VOID  
NuSocket::FD_Reset  
(  
    FD_SET* pSet  
)  
const;
```

Used to reset a bit within an `FD_SET` structure corresponding to this socket. Please see the example for `FD_Check` for more information.

Overview

Condition	Description
Pre-condition	The derived <code>NuSocket</code> object has been constructed and <code>Create</code> has been called.
Action	Calls the <code>NU_FD_Reset</code> service, which resets the bit corresponding to this socket in the bit mask designated by <code>pSet</code> .
Post-condition	The bit within <code>pSet</code> that corresponds to our socket has been reset.

Parameters

Parameter	Overview
<code>pSet</code>	A pointer to an <code>FD_SET</code> structure.

Return Value

None



Example

```
// Also see the examples for FD_Check and WaitForData

FD_SET readdata;

NU_FD_Init( &readdata );
mysocket->FD_Set( &readdata );
while ( !done )
{
    status = NU_Select(NSOCKETS,&readdata,NULL,NULL,1000);
    if (status == NU_SUCCESS)
    {
        if (mysocket->FD_Check( &readdata ))
        {
            // we got data on our socket

            // reset the mask
            mysocket->FD_Reset( &readdata );
        }
    }
}
```



NuSocket::FD_Set

```
inline
VOID
NuSocket::FD_Set
(
    FD_SET* pSet
)
const;
```

Sets the bit in a `FD_Set` structure corresponding to this socket. Please see `FD_Check` for a more thorough explanation of what this is used for.

Overview

Condition	Description
Pre-condition	has been called. The <code>NuSocket</code> derived object has been constructed and <code>Create</code> .
Action	Calls the <code>NU_FD_Set</code> service.
Post-condition	The bit corresponding to this socket has been set.

Parameters

Parameter	Overview
<code>pSet</code>	A pointer to the <code>FD_SET</code> structure in which to set the bit corresponding to this socket.

Return Value

None

Example

```
// See FD_Check for a more thorough example.
FD_SET readset;

NuUdpSocket mySocket;
mySocket.Create();

// set the bit corresponding to mySocket in the
// readset structure.
mySocket.FD_Set( &readset );
```



NuSocket::GetMulticastTTL

```
inline
STATUS
NuSocket::GetMulticastTTL
(
    UINT8& hops
)
const;
```

Returns Time To Live for multicast packets sent on this socket. This number is the number of networks that the multicast packet sent on this socket will traverse. The default value is 1, which means that most multicast packets will be transmitted on the local network only and not make it past the first router.

Overview

Condition	Description
Pre-condition	Nucleus C++ NET has been initialized, this socket has been constructed and <code>Create</code> has been called.
Action	Calls the <code>Getsockopt</code> member.
Post-condition	If successful, the hops parameter has been modified to contain the current TTL value.

Parameters

Parameter	Overview
<code>hops</code>	A reference to the 8 bit number to place the TTL value in.

Return Value

Return Value	Overview
<code>NU_INVALID_SOCKET</code>	This socket did not have a valid socket descriptor. (Either it was corrupted or <code>Create</code> wasn't called.)
<code>NU_NOT_CONNECTED</code>	This socket is not currently connected.
<code>NU_SUCCESS</code>	Indicates successful completion of the service.

Example

```
MyUdpSocket socket;
UINT8 hops;
if (socket.Create())
{
    if ( socket.GetMulticastTTL(hops) == NU_SUCCESS )
    {
        socket.SetMulticastTTL( hops + 1 );
    }
}
```



NuSocket::GetPeerAddress

```
inline
STATUS
NuSocket::GetPeerAddress
(
    SocketAddress* pAddress
)
const;
```

Called to get our peer's IP address and port number from the network stack. This overload of `GetPeerAddress` takes a pointer to a `SocketAddress` object.

Overview

Condition	Description
Pre-condition	See <code>GetPeerAddress(SocketAddress&)</code> .
Action	Calls <code>GetPeerAddress(SocketAddress&)</code> .
Post-condition	See <code>GetPeerAddress(SocketAddress&)</code> .

Parameters

Parameter	Overview
<code>pAddress</code>	A valid pointer to a <code>SocketAddress</code> object.

Return Value

Return Value	Overview
<code>NU_BAD_SOCKETD</code>	The socket descriptor for this instance is invalid.
<code>NU_NOT_CONNECTED</code>	This socket is not currently connected.
<code>NU_SUCCESS</code>	Indicates successful completion of the service.

Example

```
STATUS status;
SocketAddress* pTheirAddress = new SocketAddress;

status = pSocket->GetPeerAddress( pTheirAddress );

if (status == NU_SUCCESS)
{
    ...
}
```



NuSocket::GetPeerAddress

```

STATUS*
NuSocket::GetPeerAddress
(
    SocketAddress& address
)
const;

```

Called to get our peer's IP address and port number from the network stack. Often called by a server to obtain information on the client that they are connected to.

Overview

Condition	Description
Pre-condition	Nucleus C++ NET has been initialized. The NuSocket derived object has been constructed and Create has been called. This socket is connected to a remote peer.
Action	Calls the <code>NU_Get_Peer_Name</code> service.
Post-condition	It should be noted that this routine does not resolve the name of the peer, only the IP address and port number. If the name is needed, get the IP address with this call, and then call the API function 'NuResolveByIP'.

Parameters

Parameter	Overview
address	A reference to a SocketAddress object to copy the information into.

Return Value

Return Value	Overview
NU_BAD_SOCKETD	The socket descriptor for this instance is invalid.
NU_INVALID_PARM	One of the parameters was bad.
NU_NOT_CONNECTED	This socket is not currently connected.
NU_SUCCESS	Indicates successful completion of the service.

Example

```

SocketAddress mypeer;
...
if (serversocket->GetPeerAddress( mypeer ) == NU_SUCCESS)
{
    UINT16 port = mypeer.GetPort();
}

```



NuSocket::GetSocketDescriptor

```
inline  
SOCKETD_TYPE  
NuSocket::GetSocketDescriptor()  
const;
```

Gives access to the underlying socket descriptor for this `NuSocket` instance. This can be used when interfacing Nucleus Net C++ objects with straight stick C networking component or service call that wants a socket descriptor.

Overview

Condition	Description
Pre-condition	The object exists.
Action	Returns the socket descriptor.
Post-condition	If the socket object has a good socket descriptor, 0<= socket_descriptor < NSOCKETS. Negative if it is invalid.

Parameters

None

Return Value

Return Value	Overview
value <0	The socket descriptor for this socket is invalid.
value >=0	The function succeeded, and the value contains the socket descriptor.

Example

```
// const NuUdpSocket& the_socket ...  
  
INT socketd = the_socket.GetSocketDescriptor();
```



NuSocket::GetSocketFor

```
static
NuSocket*
NuSocket::GetSocketFor
(
    SOCKETD_TYPE nSocketDesc
);
```

Returns a pointer to the `NuSocket` instance for a given socket descriptor, `NULL` if it doesn't exist.

Overview

Condition	Description
Pre-condition	The Nucleus C++ NET component has been initialized.
Action	Returns a pointer to an <code>NuSocket</code> object for a given socket descriptor if one exists. Note that it is entirely possible to have valid socket descriptors for which an <code>NuSocket</code> object doesn't exist. This would happen if you had a system that had both Nucleus Net C++ <code>NuSocket</code> objects and straight Nucleus Net C API socket calls.
Post-condition	A pointer to the <code>NuSocket</code> object for a given descriptor is returned. <code>NULL</code> if one doesn't exist or the socket descriptor is invalid.

Parameters

Parameter	Overview
<code>nSocketDesc</code>	The socket descriptor you want to find the object for. <code>SOCKETD_TYPE</code> is most often a <code>typedef</code> for <code>INT</code> .

Return Value

Return Value	Overview
<code>aValidPointer</code>	A valid pointer is returned.
<code>NULL</code>	There is no <code>NuSocket</code> derived object corresponding to the given socket descriptor or the socket descriptor is invalid.



Example

```
// create a new TCP socket
NuTcpSocket* pFirstSocket = new NuTcpSocket;
pFirstSocket->Create();

// get the new sockets socket descriptor
int16 nDesc = pFirstSocket->GetSocketDescriptor();
....
// call the static routine to look up the object for the
socket descriptor
NuSocket* pSocket = NuSocket::GetSocketFor( nDesc );
if (pSocket != pFirstSocket)
{
    // we are in trouble...
```



NuSocket::Getsockopt

```

inline
STATUS*
NuSocket::Getsockopt
(
    INT level,
    INT optname,
    VOID* optval,
    INT* optlen,
)

```

Returns status of an option for a particular socket. Currently the only supported options are for broadcasting and multicasting support on a socket. This functionality has been rolled into member functions, which are much easier to use: See `IsBroadcastingEnabled()` and `GetMulticastTTL(UINT8&)`.

Overview

Condition	Description
Pre-condition	Nucleus NET has been initialized. This socket instance has been constructed and <code>Create</code> has been called.
Action	Calls the <code>NU_Getsockopt</code> service.
Post-condition	If successful, the location pointed to by the <code>optval</code> parameter contains the desired data, and <code>optlen</code> contains the length in bytes of that data.

Parameters

Parameter	Overview
<code>level</code>	Specifies the protocol level. The only valid entries for this parameter are <code>SOL_SOCKET</code> and <code>IPPROTO_IP</code> .
<code>optname</code>	Specifies an option. Currently the only valid values for this parameter are <code>SO_BROADCAST</code> , and <code>IP_MULTICAST_TTL</code> .
<code>optval</code>	Pointer to the location where the option status can be written.
<code>optlen</code>	<code>optlen</code> should contain the size of the location pointed to by <code>optval</code> . Upon returning specifies the size in bytes of the data copied into the location pointed to by <code>optval</code> .

Return Value

Return Value	Overview
<code>NU_INVALID</code>	<code>optval</code> is a null pointer.
<code>NU_INVALID_LEVEL</code>	The value specified in the <code>level</code> parameter is invalid.
<code>NU_INVALID_OPTION</code>	The value specified in the <code>optname</code> parameter is invalid.
<code>NU_INVALID_SOCKET</code>	This socket did not have a valid socket descriptor. (either it was corrupted or <code>Create</code> wasn't called).
<code>NU_NOT_CONNECTED</code>	This socket is not currently connected.
<code>NU_SUCCESS</code>	Indicates successful completion of the service.



Example

```
// this example checks whether or not broadcasting is
// enabled on this socket. Of course you could just
// call the IsBroadcastingEnabled() member...

STATUS status;
INT     value = 0;
INT     size = sizeof(INT);

status = Getsockopt(SOL_SOCKET, SO_BROADCAST, &value, &size);
```



NuSocket::Initialize

```
static
STATUS
NuSocket::Initialize();
```

One time initialization of the Static portions of this object.

Overview

Condition	Description
Pre-condition	None
Action	Initializes the static array that maps socket descriptors to NuSocket derived object instances.
Post-condition	The static portions of NuSocket are properly initialized and derived NuSocket objects can now be created.

Parameters

None

Return Value

Return Value	Overview
NU_SUCCESS	Indicates successful completion of the service.
value <0	The operation failed and an error code was returned.

Example

```
if (NuSocket::Initialize() == NU_SUCCESS)
{
    ...
}
```



NuSocket::IsDataAvailable

```
inline  
BOOL  
NuSocket::IsDataAvailable()  
const;
```

Returns `TRUE` if there is data available for this socket. See also `WaitForData`, which allows you to specify a timeout.

Overview

Condition	Description
Pre-condition	Nucleus C++ NET has been initialized. This socket object has been constructed and <code>Create</code> has been called. An IP and port has been bound to this socket via a call to <code>Bind</code> . In the case of TCP, a connection has been established.
Action	Calls the <code>WaitForData</code> member with no suspension allowed.
Post-condition	None

Parameters

None

Return Value

Return Value	Overview
<code>FALSE</code>	Data is NOT available.
<code>TRUE</code>	Data is available.

Example

```
...  
if ( mysocket->IsDataAvailable() )  
{  
    bytes = mysocket->Recv( buffer, 1000 );  
    ...  
}
```



NuSocket::IsValid

```
inline
BOOL
NuSocket::IsValid()
const;
```

Checks if this NuSocket derived socket is valid by examining its socket descriptor.

Overview

Condition	Description
Pre-condition	This object exists.
Action	Calls the global function IsValidSocketDescriptor.
Post-condition	None

Parameters

None

Return Value

Return Value	Overview
FALSE	If not successful.
TRUE	Socket descriptor is a valid socket descriptor.

Example

```
NuTcpSocket mysocket;
mysocket.Create();

if ( mysocket.IsValidSocket() )
{
    if ( mysocket.Connect( tcpserveraddress )
        ...
```



NuSocket::NuSocket

```
NuSocket  
NuSocket::NuSocket();
```

Constructor. `NuSocket` is an abstract class (it has pure virtual members) and as such you will only create it through a derived class.

Overview

Condition	Description
Pre-condition	None
Action	Initializes the socket descriptor to <code>-1</code> .
Post-condition	The object exists.

Parameters

None

Return Value

None

Example

`NuSocket` is an abstract class, and as such is only created through derived classes.



NuSocket::operator=

```

NuSocket&
NuSocket::operator=
(
const NuSocket& sourceSocket
);

```

Assignment operator. This is not supported with this implementation, so it is protected to force a compile time error if it is used.

Overview

Condition	Description
Pre-condition	Unsupported
Action	Unsupported
Post-condition	Unsupported

Parameters

Parameter	Overview
sourceSocket	Socket to copy.

Return Value

None



NuSocket::Ping

```
static
inline
STATUS*
NuSocket::Ping
(
const IPAddress& ip,
  UINT32 timeout_ticks
);
```

Sends one ping request to the given IP address suspending for the given number of timer ticks waiting for a reply.

Overview

Condition	Description
Pre-condition	Nucleus C++ NET has been initialized.
Action	Calls the NU_Ping service.
Post-condition	None

Parameters

Parameter	Overview
ip	A reference to the IP address to ping.
timeout_ticks	The number of Nucleus timer ticks to wait for a reply. Can also be NU_NO_SUSPEND, (which wouldn't make much sense since ping is not instantaneous) or NU_SUSPEND, which might also be bad since IP networks aren't completely reliable.

Return Value

Return Value	Overview
NU_MEM_ALLOC	The ping request was not sent because NET could not allocate the needed memory to complete the service.
NU_NO_ACTION	NET was unable to send the ping request. Possibly because there is no route available to the given IP address.
NU_NO_BUFFERS	No transmit buffers were available when the call was made.
NU_SUCCESS	Indicates successful completion of the service.
NU_TIMEOUT	A ping reply was not received within the timeout period.



Example

```
STATUS status;
IPAddress (200,100,50,0);

// ping destination and make sure that it is up
// and running. Suspend up to 100ms waiting
// for the reply.

status = NuSocket::Ping( myaddress,TicksFromMS(100) );

if (status == NU_SUCCESS )
{
    // YES!, WE STILL HAVE A VOICE!
```



NuSocket::SetBlocking

```
inline
STATUS
NuSocket::SetBlocking
(
    BOOL bBlock
);
```

Called to change the suspension mode for this socket. By default, calling `Recv`, `RecvFrom`, and `Accept` will suspend the calling thread until data is available (in the case of `Recv` and `RecvFrom`) or until a client connects (in the case of `Accept`).

Overview

Condition	Description
Pre-condition	Nucleus C++ NET has been initialized. An <code>NuSocket</code> derived object has been constructed and <code>Create</code> has been called on it.
Action	Calls the <code>NU_Fcntl</code> service.
Post-condition	If successful, the blocking mode for this socket has been changed.

Parameters

Parameter	Overview
<code>bBlock</code>	If <code>bBlock</code> is <code>TRUE</code> , then in subsequent calls to <code>Recv</code> , for example, the calling thread will suspend if no data is available. <code>FALSE</code> turns blocking off.

Return Value

Return Value	Overview
<code>NU_INVALID_SOCKET</code>	This socket did not have a valid socket descriptor. (Either it was corrupted or <code>Create</code> wasn't called.)
<code>NU_SUCCESS</code>	Indicates successful completion of the service.

Example

```
// Assume that NuTcpSocket* pSocket has been
// constructed, create has been called, and a
// connection has been established.

pSocket->SetBlocking( TRUE ); // turn blocking on
// suspend until data comes in
status = pSocket->Recv( buffer, 100 );
pSocket->SetBlocking( FALSE ); // turn blocking off
...
```



NuSocket::SetSocketDesc

```

VOID
NuSocket::SetSocketDesc
(
    SOCKETD_TYPE socket_desc
);

```

Sets the internal socket descriptor member and the corresponding static lookup table data member.

Overview

Condition	Description
Pre-condition	Nucleus C++ NET has been initialized.
Action	The Nucleus NET stack does the job of insuring that socket descriptors are unique, and thus we don't have to worry about reentrancy in access to the static table of sockets.
Post-condition	The <code>socketd</code> member has been set, and the global table of socket descriptors to instances has been updated.

Parameters

Parameter	Overview
<code>socket_desc</code>	The new socket descriptor for the <code>NuSocket</code> object.

Return Value

None

Example

```

BOOL
NuSocket::CreateBase( INT16 type )
{
    socketd = (SOCKETD_TYPE)NU_Socket( NU_FAMILY_IP,
                                       type, NU_NONE );

    SetSocketDesc( socketd );

    return ( IsValid() );
}

```



NuSocket::Setsockopt

```
inline
STATUS
NuSocket::Setsockopt
(
    INT level,
    INT optname,
    VOID* optval,
    INT optlen,
)
```

Sets socket options for this particular socket. Currently the only supported options are for toggling broadcast and multicast support on a socket. Other options will be added in the future as needed. The ability to broadcast is enabled by default. Nucleus C++ NET provides other members that set these same options in a more intuitive way, this member exists for Nucleus compatibility only. Please see:

```
NuSocket::SetMulticastTTL(),
NuSocket::JoinMulticastGroup(),
NuSocket::EnableBroadcasting(),
NuRawSocket::SetAppLayerDoesIPHeader()
```

Overview

Condition	Description
Pre-condition	Nucleus C++ NET has been initialized. This NuSocket derived socket has been constructed and Create has been called.
Action	Calls the NU_Setsockopt service.
Post-condition	If successful, the given option has been changed for this socket.

Parameters

Parameter	Overview
level	Specifies the protocol level. The only valid entries for this parameter are SOL_SOCKET, and IPPROTO_IP.
optname	Specifies an option. Valid options are SO_BROADCAST, IP_ADD_MEMBERSHIP, IP_DROP_MEMBERSHIP, IP_MULTICAST_TTL, and IP_HDRINCL.
optval	A pointer to the new value for the option.
optlen	Specifies the size in bytes of the location pointed to by optval.



Return Value

Return Value	Overview
NU_INVALID	Either the <code>optval</code> or <code>optlen</code> parameter has problems.
NU_INVALID_LEVEL	The value specified in the <code>level</code> parameter is invalid.
NU_INVALID_OPTION	The value specified in the <code>optname</code> parameter is invalid.
NU_INVALID_SOCKET	This socket did not have a valid socket descriptor. (either it was corrupted or <code>Create</code> wasn't called.)
NU_SUCCESS	Indicates successful completion of the service.

Example

```
// in this example we manually join a multicast
// group. We could just call JoinMulticastGroup,
// which is much easier, and hides all of these
// details from us...

IP_MREQ mgroup; // multicast request structure
UINT8 device_ip[4] = {200,100,50,0};
UINT8 multicast_ip[4] = {200,100,50,0}; // all systems

NuUdpSocket udpsocket;

udpsocket.Create();

// copy the IP addresses into the request structure
memcpy (&mgroup.sck_multiaddr,multicast_ip, 4);
memcpy (&mgroup.sck_inaddr,device_ip, 4);
optlen
);

udpsocket.Setsockopt(IPPROTO_IP, IP_ADD_MEMBERSHIP, &mgroup,
sizeof(mgroup));
```



NuSocket::WaitForData

```
STATUS
NuSocket::WaitForData
(
    UNSIGNED uTicksToSuspend
)
const;
```

Suspends calling thread until data available or a client is connecting with a timeout. The `Recv` and `RecvFrom` functions only have the capability to unconditionally suspend until data is available. If you need to be able to optionally timeout if data is not available after a certain amount of time, use this function. If you want to do an immediate test to see if data is available for a socket, you can also use `IsDataAvailable`. If you want to suspend waiting for data on multiple sockets, you must call the `NU_Select` routine directly. Please see the example for the `FD_Check` member.

Overview

Condition	Description
Pre-condition	Nucleus C++ NET has been initialized. This <code>NuSocket</code> derived object has been constructed and <code>Create</code> has been called. A valid <code>SocketAddress</code> must have been bound to the socket before calling this routine.
Action	Calls the <code>NU_Select</code> service. The calling thread is suspended until data is available or the ticks specified have elapsed.
Post-condition	If successful, data is available to be read from the socket or a client is attempting to connect. If a timeout occurs, <code>NU_NO_DATA</code> is returned.

Parameters

Parameter	Overview
<code>uTicksToSuspend</code>	The number of Nucleus timer ticks to suspend waiting for data or a client connection.

Return Value

Return Value	Overview
<code>NU_INVALID_SOCKET</code>	This socket did not have a valid socket descriptor. (either it was corrupted or <code>Create</code> wasn't called.)
<code>NU_NO_DATA</code>	Timed out without getting data.
<code>NU_NO_SOCKETS</code>	The socket descriptor for this object is invalid. If the object is used correctly this will not happen.
<code>NU_SUCCESS</code>	Indicates successful completion of the service.



Example

```

// This example uses WaitForData for a server to
// timeout and do background work while it is
// waiting for clients to connect. Once a client is
// connected, it uses WaitForData to wait to receive
// data from the client, and Aborts the connection if
// it times out.

STATUS stat;
NuTcpSocket* serverSocket = new NuTcpSocket;
NuTcpSocket* worker = new NuTcpSocket;

serverSocket->Create();
serverSocket->Bind( serverAddress );
serverSocket->Listen( 10 );

while( 1 )
{
    // suspend the server for up to 1000 ticks
    // waiting for a client to connect
    stat = serverSocket->WaitForData( 1000 );

    if (stat == NU_SUCCESS)
    {
        // yes, a client is attempting a connection
        // create a worker socket to handle it.
        worker->Create();

        // accept the connection from the client
        if (serverSocket.Accept( worker ) >= 0)
        {
            // a client has connected. Wait for up to
            // 500 ticks for them to send us some data

            stat = worker->WaitForData( 500 );
            if (stat == NU_SUCCESS)
            {
                bytes = worker->Recv( buffer, 1000 );
                ...
            }
            else
            {
                // we timed out or the connection
                // was closed or something
                worker->Abort();
            }
        }
        // end of accept
        worker->Close();
    }
    else
    if (stat == NU_NO_DATA)
    {
        // we timed out waiting for a client
        // to connect to us.
        Do_Background_Stuff();
    }
}
} //end while(1)

```



class NuTcpSocket : public NuSocket

NuTcpSocket is a derived NuSocket class that specifies in its default Constructor that it is a TCP connection. Where an NuSocket instance can be either TCP/UDP/or RAW, this derived version can only be a TCP/IP socket.

Public Member Functions

Member	Overview
~NuTcpSocket	Destructor
Accept	An overload of Accept that only takes pointers to the new socket and client address objects.
Accept	Called by a server to accept a connection from a client.
Connect	Called by a client to establish a connection with a server.
Connect	Called by a client to connect to a server.
Create	Allocates space in the network stack for this socket.
EnableTcpDelay	Enable/Disable use of the Nagle algorithm for this socket.
IsConnected	Returns TRUE if a connection has been established.
Listen	Called by a server to indicate that it is willing to accept connection requests from clients.
NuTcpSocket	Default Constructor
Recv	Connection based receive.
Send	Connection based send.



NuTcpSocket::~NuTcpSocket

```
virtual  
VOID*  
NuTcpSocket::~NuTcpSocket();
```

Destructor

Overview

Condition	Description
Pre-condition	None
Action	None
Post-condition	The object doesn't exist.

Parameters

None

Return Value

None

Example

Destructors are called automatically when the object goes out of scope or is deleted.



NuTcpSocket::Accept

```
inline
STATUS
NuTcpSocket::Accept
(
    NuTcpSocket* pNewSocket,
    SocketAddress* pPeerAddress
)
```

An overload of `Accept` that only takes pointers to the new socket and client address objects. See: `Accept (NuTcpSocket&, SocketAddress&)` for more information.

Overview

Condition	Description
Pre-condition	See <code>Accept (NuTcpSocket&, SocketAddress&)</code> .
Action	Calls the <code>Accept (NuTcpSocket&, SocketAddress&)</code> overload.
Post-condition	see <code>Accept (NuTcpSocket&, SocketAddress&)</code> .

Parameters

Parameter	Overview
<code>pNewSocket</code>	A pointer to the <code>NuTcpSocket</code> instance that is to receive the incoming client connection. No check for <code>NULL</code> is done.
<code>pPeerAddress</code>	A valid pointer to a <code>SocketAddress</code> object to be filled out with the connecting client's IP address and port number.



Return Value

Return Value	Overview
NU_INVALID_PROTOCOL	The combination of protocol family and type of socket mapped to protocol that is not supported. (i.e. calling a tcp specific function on a udp socket)
NU_INVALID_SOCKET	This socket did not have a valid socket descriptor. (either it was corrupted or Create wasn't called)
NU_NO_PORT_NUMBER	No port number was stored in the socket descriptor.
NU_NO_SOCKET_MEMORY	There was not enough memory available to the network stack to allocate a new socket descriptor structure.
NU_NO_SOCKET_SPACE	There are no available sockets.
NU_NO_TASK_MATCH	The user specified non-blocking and no connection is available.
value = 0	The returned value is the Nucleus NET socket descriptor for the new socket. (0 <= socket_descriptor < NSOCKETS) if successful. pPeerAddress has been filled out with the IP address and port of the connected client, and pNewSocket has been configured for the given connection.

Example

```

SocketAddress clientAddress;
...
pWorkerSocket = new NuTcpSocket;

if ( serverSocket.Accept(pWorkerSocket, &clientAddress) >= 0 )
{
    // worker socket has now been created and is ready...
}

```



NuTcpSocket::Accept

```
STATUS  
NuTcpSocket::Accept  
(  
    NuTcpSocket& newSocket,  
    SocketAddress& peerAddress  
);
```

Called by a server to accept a connection from a client. This call can function in one of two ways: Blocking and non blocking. If blocking is turned on, then this call will suspend the calling thread until a client connection is established. Otherwise it returns with an appropriate error code.

Overview

Condition	Description
Pre-condition	Nucleus C++ NET has been initialized. <code>Create</code> , <code>Bind</code> , and <code>Listen</code> have been called on the server socket. <code>Create</code> should NOT be called on the <code>newSocket</code> as the network stack does this for you.
Action	Calls the <code>NU_Accept</code> service.
Post-condition	None

Parameters

Parameter	Overview
<code>newSocket</code>	Contains a reference to the <code>NuTcpSocket</code> object which is to accept the connection.
<code>peerAddress</code>	A reference to a <code>SocketAddress</code> object that will receive the remote connections address information once connected.



Return Value

Return Value	Overview
NU_INVALID_PROTOCOL	The combination of protocol family and type of socket mapped to protocol that is not supported. (i.e. calling a tcp specific function on a udp socket)
NU_INVALID_SOCKET	This socket did not have a valid socket descriptor. (either it was corrupted or Create wasn't called)
NU_NO_PORT_NUMBER	No port number was stored in the socket descriptor.
NU_NO_SOCKET_SPACE	There are no available sockets.
NU_NO_TASK_MATCH	The user specified non-blocking and no connection is available.
value >= 0	The returned value is the Nucleus NET socket descriptor for the new socket. (0 <= socket_descriptor < NSOCKETS) if successful. pPeerAddress has been filled out with the IP address and port of the connected client, and pNewSocket has been configured for the given connection.

Example

```
// assume that pServerSocket is a NuTcpSocket object
// that has been new'ed and Create has been called.

SocketAddress clientaddress;
TcpSocket      workerSocket;
...
workerSocket.Create();
if ( pServerSocket->Accept( workerSocket, clientaddress ) >=
0)
{
    ...
}
```



NuTcpSocket::Connect

```
inline
STATUS
NuTcpSocket::Connect
(
const SocketAddress& serveraddress
);
```

Called by a client to establish a connection with a server. An overload that takes a reference for the serverAddress.

Overview

Condition	Description
Pre-condition	Nucleus C++ NET has been initialized. Create has been called for this NuTcpSocket.
Action	Calls the NU_Connect service.
Post-condition	If successful, the connection exists and the socket is ready to send/receive data. A socket descriptor with a value >= zero is returned. If unsuccessful, an error code is returned.

Parameters

Parameter	Overview
serveraddress	A reference to a SocketAddress object containing the IP address and port of the server to connect to.



Return Value

Return Value	Overview
NU_INVALID_PARM	One of the parameters was bad.
NU_INVALID_SOCKET	This socket did not have a valid socket descriptor. (Either it was corrupted or <code>Create</code> wasn't called.)
NU_NO_HOST_NAME	The name of the host was not specified (used for look-up in the machine table set up from the Nucleus NET <code>defmachinfo</code> structure) and no other address (IP and port) values were specified.
NU_NO_PORT_NUMBER	Was not able to find an available port.
NU_NO SOCK_MEMORY	There was not enough memory available to the networking stack to allocate the structures necessary for a connection.
NU_NOT_CONNECTED	The connection attempt failed.
value >= 0	The Nucleus NET socket descriptor for the new socket is returned (<code>0 <= socket_descriptor < NSOCKETS</code>).

Example

```

SocketAddress myserver;
myserver.SetName("MYSERVER");
myserver.SetFamily( NU_FAMILY_IP );
myserver.SetPort( 80 );
myserver.SetIP( 200, 100, 50, 0 );
if ( pSocket->Connect( myserver ) >= 0)
{
    ...
}

```

NuTcpSocket::Connect

```
inline
STATUS
NuTcpSocket::Connect
(
const SocketAddress* pServerAddress = NULL
);
```

Called by a client to establish a connection with a server. Overload that takes a pointer.

Overview

Condition	Description
Pre-condition	Nucleus C++ NET has been initialized, and the <code>Create</code> member has been called for this <code>NuTcpSocket</code> instance.
Action	Calls the <code>Connect (const SocketAddress&)</code> overload.
Post-condition	If successful, the connection exists and the socket is ready to send/receive data. A socket descriptor with a value <code>>= zero</code> is returned. If unsuccessful, an error code is returned.

Parameters

Parameter	Overview
<code>pServerAddress</code>	A valid pointer to a <code>SocketAddress</code> object containing the IP address and port of the server to connect to.



Return Value

Return Value	Overview
NU_INVALID_PARM	One of the parameters was bad.
NU_INVALID_SOCKET	This socket did not have a valid socket descriptor. (either it was corrupted or <code>Create</code> wasn't called)
NU_NO_HOST_NAME	The name of the host was not specified (used for look-up in the machine table set up from the Nucleus NET <code>defmachinfo</code> structure) and no other address (IP and port) values were specified.
NU_NO_PORT_NUMBER	Was not able to find an available port.
NU_NO SOCK_MEMORY	There was not enough memory available to the networking stack to allocate the structures necessary for a connection.
value >= 0	The Nucleus NET socket descriptor for the new socket is returned. (0 <= socket_descriptor < NSOCKETS).

Example

```

SocketAddress* pAddress = new SocketAddress;
NuTcpSocket* pSocket = new NuTcpSocket;
pSocket->Create();

pAddress->SetIP( 200, 100, 50, 0 );
pAddress->SetPort( 80 );

// use default argument (NULL) to indicate that we
// want to connect using the internal remoteaddress
// member
pSocket->Connect(pAddress);

// check to see if we succeeded by calling IsValid
if (pSocket->IsValid())
{
    ...

    ...
}

```



NuTcpSocket::Create

```
virtual  
BOOL  
NuTcpSocket::Create();
```

Create is used for explicit creation of a socket after the Constructor has been called. It allocates space in the network stack for this instance of an NuTcpSocket object. Creation of a socket is then always a two part sequence, First constructing the socket, then calling Create to initialize it. Create can be called again for the socket AFTER a call to Close() or Abort(). In this way, you "re-use" the socket object without having to allocate and deallocate

Overview

Condition	Description
Pre-condition	The object has been constructed.
Action	Allocates a new socket descriptor from Nucleus NET for this instance by calling NuSocket::CreateBase.
Post-condition	If successful, space for this socket has been allocated in the networking stack and TRUE is returned.

Parameters

None

Return Value

Return Value	Overview
FALSE	If not successful.
TRUE	If successful.



Example

```
// create socket on the stack
NuTcpSocket workersocket;
SocketAddress dest;

// dest.SetIP( 200, 100, 50, 0 );
dest.SetPort( 80 );

while( 1 )
{
    // allocate space in the network
    stackworkersocket.Create();
    if (workersocket.Connect(dest) == NU_SUCCESS)

        // deallocate the socket from the network stack
        workersocket.Close();
}
```



NuTcpSocket::EnableTcpDelay

```
inline
STATUS
NuTcpSocket::EnableTcpDelay
(
    BOOL bOn
);
```

Called to control whether or not the Nagle algorithm is used for this socket. The Nagle Algorithm, is a means for coalescing many small packets into a single large packet. It delays the transmittal of a small data packet for a brief (1/4 second by default) period of time, in the expectation that the application will have another small packet to send which can be combined with the current one. When sending large data packets it does not have an impact as they will be transmitted instantly.

Overview

Condition	Description
Pre-condition	Nucleus C++ NET has been initialized. The NuTcpSocket object has been constructed and Create has been called.
Action	Calls the Setsockopt member.
Post-condition	If successful, the Nagle option for this socket has been changed.

Parameters

Parameter	Overview
bOn	If bOn == TRUE then the Nagle algorithm is turned on. If bOn == FALSE, it is turned off.

Return Value

Return Value	Overview
NU_INVALID_SOCKET	This socket did not have a valid socket descriptor, (either it was corrupted or Create wasn't called).
NU_SUCCESS	Indicates successful completion of the service.



Example

```
NuTcpSocket myClientSocket;

myClientSocket.Create();

myClientSocket.Connect( theServersAddress );

// turn the delay off so that packets are sent
// immediately.
myClientSocket.EnableTcpDelay( FALSE );

while( !done )
{
    myClientSocket.Send( data, length );
    len = myClientSocket.Recv( data, MAX_LENGTH );
    if( HandleResponse( data, len ) == QUIT)
        done = TRUE;
}
```



NuTcpSocket::IsConnected

```
inline  
BOOL  
NuTcpSocket::IsConnected()  
const;
```

This function determines if a connection has been established for this instance.

Overview

Condition	Description
Pre-condition	Nucleus C++ NET has been initialized, this object has been constructed and <code>Create</code> has been called.
Action	Calls the <code>NU_Is_Connected</code> service.
Post-condition	None

Parameters

None

Return Value

Return Value	Overview
FALSE	If NOT connected.
TRUE	If connected.

Example

```
if ( !mysocket.IsConnected() )  
{  
    // oh dear, we lost our connection...  
}
```



NuTcpSocket::Listen

```

inline
STATUS
NuTcpSocket::Listen
(
    UINT16 backlog
);

```

Called by a server to indicate that it is willing to accept connection requests from clients.

Overview

Condition	Description
Pre-condition	Nucleus C++ NET has been initialized. <code>Create</code> has been called for this <code>NuTcpSocket</code> instance.
Action	Calls the <code>NU_Listen</code> service to mark the socket as ready to accept connections.
Post-condition	The server socket is ready to accept connections from clients via a call to <code>Accept</code> .

Parameters

Parameter	Overview
<code>backlog</code>	Contains the number of connection requests that can be queued for the server.

Return Value

Return Value	Overview
<code>NU_INVALID_SOCKET</code>	This socket did not have a valid socket descriptor. (either it was corrupted or <code>Create</code> wasn't called).
<code>NU_NO SOCK_MEMORY</code>	There was not enough memory available to the networking stack to allocate the structures necessary for a connection.
<code>NU_NOT_A_TASK</code>	This routine must be called from a Task thread. It was probably called from an interrupt thread that did not perform the proper context save operation.
<code>NU_SUCCESS</code>	Indicates successful completion of the service.



Example

```
STATUS status;
NuTcpSocket serverSocket;
SocketAddress serverAddress;
...
if ( serverSocket.Create() )
{
    serverSocket.Bind(serverAddress);

    // allow 10 requests to be queued
    if (NU_SUCCESS == serverSocket.Listen( 10 ))
    {
        ...
    }
}
```



NuTcpSocket::NuTcpSocket

NuTcpSocket
NuTcpSocket::NuTcpSocket();

Constructor. Constructs a TCP specific NuSocket derived object. Must be used in conjunction with the Create function.

Overview

Condition	Description
Pre-condition	None
Action	Initializes the NuTcpSocket object. Does NOT do anything in the TCP/IP stack, which means that this object can be created anywhere you please, even before Nucleus NET is initialized.
Post-condition	The object exists.

Parameters

None

Return Value

None

Example

```
// construct the object
NuTcpSocket* pSocket = new NuTcpSocket;

// do the actual work of allocating space in the
// TCP/IP stack for this socket object
pSocket->Create();

if ( pSocket->IsValidSocket() )
{
    pSocket->Connect( &myserver );
    ...
}
else
{
    cry_hard();
}
```



NuTcpSocket::Recv

```
inline
INT32
NuTcpSocket::Recv
(
    CHAR* pData,
    UINT16 nbytes
);
```

Connection based receive.

Overview

Condition	Description
Pre-condition	Nucleus C++ NET has been initialized. Create has been called for this socket object, and a connection is in place.
Action	Calls the NU_Recv service. Note that this can suspend the caller.
Post-condition	If successful, data has been copied into the data buffer and the number of bytes received is returned. If unsuccessful, an error code has been returned.

Parameters

Parameter	Overview
pData	A pointer to the buffer to place received data in.
nbytes	The maximum number of bytes that can be placed in the data buffer.

Return Value

Return Value	Overview
NU_INVALID_SOCKET	This socket did not have a valid socket descriptor. (either it was corrupted or Create wasn't called).
NU_NO_DATA_TRANSFER	The data transfer was not completed.
NU_NO_PORT_NUMBER	No port number was stored in the socket descriptor.
value > 0	The actual number of bytes that were transferred is returned.

Example

```
CHAR mydata[100];
INT32 actualBytes;
...
actualBytes = socket.Recv( mydata, 100 );
```



NuTcpSocket::Send

```
inline
INT32
NuTcpSocket::Send
(
    CHAR* pData,
    UINT16 nbytes
);
```

Connection based send.

Overview

Condition	Description
Pre-condition	Nucleus C++ NET has been initialized. <code>Create</code> has been called for this <code>NuTcpSocket</code> instance and a connection has been established.
Action	Calls the <code>NU_Send</code> service. Note that this can suspend the calling thread.
Post-condition	If successful, the number of bytes transferred is returned, otherwise a status code.

Parameters

Parameter	Overview
<code>pData</code>	A pointer to the data to send.
<code>nbytes</code>	The number of bytes to send.

Return Value

Return Value	Overview
<code>NU_INVALID_SOCKET</code>	This socket did not have a valid socket descriptor. (either it was corrupted or <code>Create</code> wasn't called.)
<code>NU_NO_BUFFERS</code>	No transmit buffers were available when the call was made.
<code>NU_NO_PORT_NUMBER</code>	No local port number was stored in the socket descriptor. (did you call <code>Bind</code> ?)
<code>NU_NOT_CONNECTED</code>	The data transfer was not completed. This probably occurred because the connection was closed for some reason.
<code>value > 0</code>	The actual number of bytes that were transferred is returned.

Example

```
CHAR data[100];
INT32 actualBytesSent;
...
actualBytesSent = pTcpSocket->Send( data, 100 );
```



class NuUdpSocket : public NuSocket

NuUdpSocket is a derived NuSocket class that specifies in its default Constructor that it is a UDP connection. Where an NuSocket instance can be either TCP/UDP/or RAW, this derived version can only be a UDP socket. Thus calls such as Send and Recv are not allowed.

Public Member Functions

Member	Overview
~NuUdpSocket	Destructor
Create	Allocates space in the network stack for this socket.
NuUdpSocket	Default Constructor.
RecvFrom	Connectionless receive.
RecvFrom	Connectionless receive.
SendTo	Connectionless send.
SendTo	Connectionless send.



NuUdpSocket::~NuUdpSocket

```
virtual  
VOID*  
NuUdpSocket::~NuUdpSocket();
```

Destructor

Overview

Condition	Description
Pre-condition	None
Action	None
Post-condition	The object doesn't exist.

Parameters

None

Return Value

None

Example

Destructors are called automatically when the object goes out of scope or is deleted.



NuUdpSocket::Create

```
virtual
NuUdpSocket
NuUdpSocket::Create();
```

Create is used for explicit creation of a socket after the Constructor has been called. It is used to allocate space in the network stack for this instance of NuSocket derived object. Creation of a socket is then always a two part sequence, First, constructing the socket, and then calling Create to initialize it. Create can be called again for the socket AFTER a call to Close() or Abort(). In this way, you "re-use" the socket object without having to allocate and deallocate.

Overview

Condition	Description
Pre-condition	The object has been constructed.
Action	Allocates a new socket descriptor from Nucleus NET for this instance.
Post-condition	Space in the network stack has been allocated for this socket instance. The socketd member contains the new socket descriptor.

Parameters

None

Return Value

Return Value	Overview
FALSE	Space in the network stack has NOT been allocated for this socket instance.
TRUE	Space in the network stack has been successfully allocated for this socket instance.



Example

```
NuUdpSocket  workersocket;
SocketAddress dest;

dest.SetIP( 200, 100, 50, 0 );
dest.SetPort( 7 );

while( 1 )
{
    // allocate space in the network stack
    workersocket.Create();

    workersocket.SendTo( dataptr, datasize, dest );

    // deallocate the socket from the network stack
    workersocket.Close();

    // sleep for a second and do it again
    Task::Sleep( systemClockFrequency );
}
```



NuUdpSocket : : NuUdpSocket

NuUdpSocket

NuUdpSocket : : NuUdpSocket () ;

Constructor. Constructs a UDP specific NuSocket derived object.

Overview

Condition	Description
Pre-condition	None
Action	No real work is done in this Constructor. The bulk of the creation is done in NuUdpSocket : : Create.
Post-condition	The object exists. (but still needs to be created...)

Parameters

None

Return Value

None

Example

```
SocketAddress destination;  
destination.SetIP( 200, 100, 50, 0 );  
destination.SetPort( 127 );  
  
NuUdpSocket* pSocket = new NuUdpSocket;  
  
if ( pSocket->Create() )  
{  
    pSocket->SendTo( dataptr, datasize, &destination );  
    ...  
}
```



NuUdpSocket::RecvFrom

```

inline
INT32
NuUdpSocket::RecvFrom
(
    CHAR* pData,
    UINT16 nbytes,
    SocketAddress& from
)

```

Connectionless receive. This is an overload that takes a reference to a `SocketAddress` object to write the peer sockets information into.

Overview

Condition	Description
Pre-condition	Nucleus C++ NET has been initialized. <code>Create</code> has been called for this <code>NuUdpSocket</code> instance.
	Nucleus C++ NET has been initialized. <code>Create</code> has been called for this <code>NuUdpSocket</code> instance. Either: 1) <code>Bind</code> was called to explicitly pick a port to receive from. 2) A previous call to <code>SendTo</code> was made and a port number assigned to the socket through that.
Action	Calls the <code>NU_Recv_From</code> service. This can suspend the calling thread.
Post-condition	If successful, data has been copied into the data buffer, <code>from</code> contains the IP address and port of the sender, and the number of bytes received is returned. If unsuccessful, an error code has been returned.

Parameters

Parameter	Overview
<code>pData</code>	A pointer to the data buffer to place received data in.
<code>nbytes</code>	The maximum number of received bytes that can be placed in the data buffer.
<code>from</code>	A reference to a <code>SocketAddress</code> containing the IP address and port of the socket to receive from.



Return Value

Return Value	Overview
NU_INVALID_SOCKET	This socket did not have a valid socket descriptor. (either it was corrupted or Create wasn't called)
NU_NO_DATA_TRANSFER	The data transfer was not completed.
NU_NO_PORT_NUMBER	No port number was stored in the socket descriptor.
value > 0	The actual number of bytes received is returned.

Example

```

CHAR          buffer[1500];
NuUdpSocket   socket;
SocketAddress  address( IPAddress(0,0,0,0), 10 );
INT32         bytes;

if (socket.Create())
{
    // bind to the desired port
    socket.Bind(address);

    // suspend waiting for data on the port
    bytes = socket.RecvFrom(buffer,1500,address);
}

```



NuUdpSocket::RecvFrom

```

inline
INT32
NuUdpSocket::RecvFrom
(
    CHAR* pData,
    UINT16 nbytes,
    SocketAddress* pFrom
)

```

Connectionless receive. This is an overload that takes a pointer to a `SocketAddress` object to write the peer sockets information into.

Overview

Condition	Description
Pre-condition	See <code>RecvFrom(CHAR*, UINT16, SocketAddress&)</code> .
Action	Calls the <code>RecvFrom(CHAR*, UINT16, SocketAddress&)</code> overload.
Post-condition	If successful, data has been copied into the data buffer and the number of bytes received is returned. The IP unsuccessful, an error code has been returned

Parameters

Parameter	Overview
<code>pData</code>	Pointer to a data buffer to place received data into.
<code>nbytes</code>	The maximum number of bytes that the receive buffer can contain.
<code>pFrom</code>	A valid pointer to a <code>SocketAddress</code> object to write the peer sockets IP address and port information into.

Return Value

Return Value	Overview
<code>NU_INVALID_SOCKET</code>	This socket did not have a valid socket descriptor. (either it was corrupted or <code>Create</code> wasn't called).
<code>NU_NO_DATA_TRANSFER</code>	The data transfer was not completed.
<code>NU_NO_PORT_NUMBER</code>	No port number was stored in the socket descriptor.
<code>value > 0</code>	The actual number of bytes received is returned.



Example

```
SocketAddress  peerAddr;

NuUdpSocket   socket;
CHAR          buffer[1500];
INT32         bytes;

if ( socket.Create() )
{
    peerAddr.SetIP(200,100,50,0 );
    peerAddr.SetPort( 7 );

    FillBufferWithSomeCommand( buffer );

    socket.SendTo( buffer, 1500, peerAddr );

    // suspend up to 20 ticks waiting for a
    // response.
    if (socket.WaitForData( 20 ) == NU_SUCCESS)
    {
        bytes = socket.RecvFrom(buffer,1500,peerAddr);

        // handle the response
        HandleResponse( buffer, bytes );
    }
};
else
{
    // you timed out waiting for data
    ...
}
}
```



NuUdpSocket::SendTo

```

inline
INT32
NuUdpSocket::SendTo
(
    CHAR* pData,
    UINT16 nbytes,
    SocketAddress& dest
);

```

Connectionless send. This overload takes a reference for the destination `SocketAddress`.

Overview

Condition	Description
Pre-condition	Nucleus C++ NET has been initialized. <code>Create</code> has been called for this <code>NuUdpSocket</code> .
Action	Calls the <code>NU_Send_To</code> service routine. The calling thread may be suspended.
Post-condition	If a port number was not explicitly specified via a call to <code>Bind</code> , an open port was obtained automatically in the call to <code>SendTo</code> . This port will be used in subsequent calls to both <code>SendTo</code> and <code>Recv</code> until the socket is closed. The <code>dest</code> parameter is unmodified.

Parameters

Parameter	Overview
<code>pData</code>	A pointer to the data buffer to send.
<code>nbytes</code>	The number of bytes to send.
<code>dest</code>	A reference to a <code>SocketAddress</code> containing the IP address and port number of the socket to send to.

Return Value

Return Value	Overview
<code>NU_INVALID_ADDRESS</code>	The address passed in was most likely incomplete (i.e., missing the IP number).
<code>NU_INVALID_SOCKET</code>	This socket did not have a valid socket descriptor. (either it was corrupted or <code>Create</code> wasn't called).
<code>NU_NO_DATA_TRANSFER</code>	The data transfer was not completed.
<code>NU_NO_PORT_NUMBER</code>	No port number was stored in the socket descriptor.
<code>value > 0</code>	The actual number of bytes that were transferred is returned.



Example

```
NuUdpSocket    socket;
SocketAddress dest;
SocketAddress responseAddress;
CHAR           buffer[1500];
INT32          bytes;

if ( socket.Create() )
{
    dest.SetIP(200,100,50,0 );
    dest.SetPort( 7 );
    FillBufferWithSomeCommand( buffer );
    socket.SendTo( buffer, 1500, dest );

    // suspend waiting for a response
    bytes = socket.RecvFrom( buffer, 1500,
                           responseAddress );

    // handle the response
    HandleResponse( buffer, bytes );
}
```



NuUdpSocket::SendTo

```

inline
INT32
NuUdpSocket::SendTo
(
    CHAR* pData,
    UINT16 nbytes,
    SocketAddress* pDest
);

```

Connectionless send. This overload takes a pointer to the `SocketAddress` to send to.

Overview

Condition	Description
Pre-condition	See <code>SendTo (CHAR*,UINT16,SocketAddress&)</code> .
Action	Calls the <code>SendTo (CHAR*,UINT16,SocketAddress&)</code> overload.
Post-condition	See <code>SendTo (CHAR*,UINT16,SocketAddress&)</code> .

Parameters

Parameter	Overview
<code>pData</code>	A pointer to the data buffer to send.
<code>nbytes</code>	The number of bytes in the buffer to send.
<code>pDest</code>	A valid pointer to a <code>SocketAddress</code> object containing the IP address and port of the socket to send to.

Return Value

Return Value	Overview
<code>NU_INVALID_ADDRESS</code>	The address passed in was most likely incomplete(i.e., missing the IP number).
<code>NU_INVALID_SOCKET</code>	This socket did not have a valid socket descriptor. (Either it was corrupted or <code>Create</code> wasn't called).
<code>NU_NO_DATA_TRANSFER</code>	The data transfer was not completed.
<code>NU_NO_PORT_NUMBER</code>	No port number was stored in the socket descriptor.
<code>value > 0</code>	The actual number of bytes that were transferred is returned.



Example

```
SocketAddress destAddr;
NuUdpSocket  socket;
CHAR         buffer[1500];
INT32        bytes;

if ( socket.Create() )
{
    destAddr.SetIP(200,100,50,0 );
    destAddr.SetPort( 7 );

    FillBufferWithSomeCommand( buffer );

    socket.SendTo( buffer, 1500, &destAddr );

    // suspend waiting for a response
    bytes = socket.RecvFrom( buffer, 1500, &destAddr);

    // handle the response
    HandleResponse( buffer, bytes );
}
```



class SocketAddress

An object used in a number of NuSocket members as well as supporting functions. It contains information not found in a raw IP address such as port number and a the text name corresponding to the IP address if it exists. It is used in a number of the Supporting functions as well as NuSocket members. It contains additional information not found in a raw IP address such as port number and associated name. It encapsulates the net address structure, which contains an IP address, port number, associated name, and networking family specification.

Public Member Functions

Member	Overview
~SocketAddress	Destructor
GetFamily	Returns Family.
GetIP	Returns an IPAddress object.
GetIPBits	Returns the IP address as a raw 32 bit unsigned value.
GetName	Returns a pointer to the name.
GetPort	Returns the port number.
operator =	Assignment from the "C" structure.
operator =	Assignment operator.
operator const struct addr_struct*	Const casting operator for "C" API compatibility.
operator struct addr_struct*	Casting operator for "C" API compatibility.
Set	Sets IP address, port number and name.
SetFamily	Sets the family.
SetIP	Sets the IP address by individual octets.
SetIP	Set the IP address by a reference to an IPAddress object.
SetIP	Sets the IP address as a 32 bit number.
SetIP	Sets the IP by copying an IPAddress object.
SetName	Sets the name pointer.
SetPort	Sets the port.
SocketAddress	Copy Constructor.
SocketAddress	"C" structure copy Constructor.
SocketAddress	Default Constructor.
SocketAddress	Constructor that takes an IP address, port and name.



SocketAddress::~~SocketAddress

```
virtual  
SocketAddress  
SocketAddress::~~SocketAddress();
```

Destructor

Overview

Condition	Description
Pre-condition	None
Action	None
Post-condition	The object does not exist.

Parameters

None

Return Value

None

Example

Destructors are called automatically when the object goes out of scope or is deleted.



SocketAddress::GetFamily

```
inline
UINT16
SocketAddress::GetFamily()
const;
```

Called to give access to the networking family of the given `SocketAddress` object. Currently the only valid value for this parameter is `NU_FAMILY_IP`, the default value, so you don't have to worry about setting it.

Overview

Condition	Description
Pre-condition	None
Action	Returns the family type as an unsigned 16 bit integer.
Post-condition	None

Parameters

None

Return Value

Return Value	Overview
<code>NU_FAMILY_IP</code>	This protocol belongs to the IP family.

Example

```
SocketAddress servaddr;
...
if (servaddr.GetFamily() != NU_FAMILY_IP)
{
    // should never happen....
}
```



SocketAddress::GetIP

```
inline  
IPAddress  
SocketAddress::GetIP()  
const;
```

Returns the IP address as an `IPAddress` object.

Careful! Returns an object on the stack. Also see member `GetIPBits`

Overview

Condition	Description
Pre-condition	None
Action	Retrieves the IP address contained in the private address member. Careful! Returns an object on the stack.
Post-condition	None

Parameters

None

Return Value

Return Value	Overview
<code>validObject</code>	A new <code>IPAddress</code> object with the contained IP address is returned.

Example

```
SocketAddress servaddr;  
IPAddress ip;  
...  
ip = servaddr.GetIP();
```



SocketAddress::GetIPBits

```
inline
UINT32
SocketAddress::GetIPBits()
const;
```

Returns the IP address as a raw 32 bit unsigned value.

Overview

Condition	Description
Pre-condition	None
Action	None
Post-condition	None

Parameters

None

Return Value

Return Value	Overview
validData	The IP address is returned as a 32 bit unsigned value.

Example

```
SocketAddress theobj;

UINT32 theip = theobj.GetIPBits();
```



SocketAddress::GetName

```
inline  
CHAR*  
SocketAddress::GetName()  
const;
```

Called to give access to the name corresponding to a given SocketAddress instance.

Overview

Condition	Description
Pre-condition	None
Action	Returns a pointer to the name.
Post-condition	None

Parameters

None

Return Value

Return Value	Overview
validString	The SocketAddress object contains a pointer to the name which was passed into it or the name as it exists inside of the DNS information stored in the stack. Thus you should not modify the data referenced by the returned pointer, which is why is a <code>const char*</code> .

Example

```
const char* pName;  
SocketAddress mysocket;  
...  
mysocket.SetName( "www.mysite.com")  
...  
pName = mysocket.GetName();
```



SocketAddress::GetPort

```
inline
UINT16
SocketAddress::GetPort()
const;
```

Returns the 16 bit port address contained in the SocketAddress object.

Overview

Condition	Description
Pre-condition	None
Action	The 16 bit port value is returned.
Post-condition	None

Parameters

None

Return Value

Return Value	Overview
validData	The 16 bit port value is returned, the object is unmodified.

Example

```
SocketAddress servaddr;
UINT16 theport;
...
theport = servaddr.GetPort();
```



SocketAddress::operator =

```
inline
SocketAddress
SocketAddress::operator =
(
const SocketAddress& newAddr
);
```

Assignment operator

Overview

Condition	Description
Pre-condition	None
Action	Copies the contents of <code>newAddr</code> into the current <code>SocketAddress</code> instance.
Post-condition	None

Parameters

Parameter	Overview
<code>newAddr</code>	A <code>SocketAddress</code> object that we wish to copy.

Return Value

Return Value	Overview
<code>aValidReference</code>	A reference to the current object.

Example

```
IPAddress ip(200,100,50,0);
SocketAddress address1(ip, 80 );
SocketAddress address2;

address2 = address1;
```



SocketAddress::operator =

```
inline
SocketAddress&
SocketAddress::operator =
(
const struct addr_struct& addr
);
```

Assignment operator that takes a "C" address structure.

Overview

Condition	Description
Pre-condition	None
Action	Does a binary copy of the structure into the internal member.
Post-condition	None

Parameters

Parameter	Overview
addr	A const reference to the "C" Nucleus NET address structure.

Return Value

Return Value	Overview
aValidReference	A reference to the current object.

Example

```
struct addr_struct  c_structure;
SocketAddress       address2;

// assign values the "C" way
c_structure.family = NU_FAMILY_IP;
c_structure.port = 80;
c_structure.id.is_ip_addrs[0] = 200;
c_structure.id.is_ip_addrs[1] = 100;
c_structure.id.is_ip_addrs[2] = 50;
c_structure.id.is_ip_addrs[3] = 0;
c_structure.name = "destination.com";

// call assignment operator
address2 = c_structure;
```



SocketAddress::operator const struct addr_struct*

```
inline
struct addr_struct*
SocketAddress::operator const struct addr_struct*()
const;
```

Casting operator to get access to our internal struct addr_struct member. This version is used when you have a const SocketAddress object.

Overview

Condition	Description
Pre-condition	None
Action	Returns a pointer to the internal member.
Post-condition	None

Parameters

None

Return Value

Return Value	Overview
aValidPointer	A const pointer to the internal data member is returned.

Example

```
IPAddress ip(200,100,50,0);
const SocketAddress myAddress(ip,80,"mycompany.com");
struct addr_struct c_structure;

// here because myAddress is const it cannot be
// modified. Fortunately, memcpy takes as a
// parameter for the source a const void* meaning
// that it will not modify the data, so we can get
// a const pointer to our object via the cast
// operator and pass it into the call to memcpy.

memcpy( &c_structure,
        (const struct addr_struct*)myAddress,
        sizeof(struct addr_struct));
```



SocketAddress::operator struct addr_struct*

```
inline
struct addr_struct*
SocketAddress::operator struct addr_struct*();
```

Casting operator to get access to our internal `struct addr_struct` member. This is here for compatibility with C API routines that expect this type. You probably won't ever need to use this function because this is taken care of internally.

Overview

Condition	Description
Pre-condition	None
Action	Returns a pointer to our internal <code>struct addr_struct</code> member.
Post-condition	None

Parameters

None

Return Value

Return Value	Overview
<code>aValidPointer</code>	A pointer to the internal <code>struct addr_struct</code> member is returned. Will never be <code>NULL</code> if this object exists.

Example

```
SocketAddress socketAddress1;
NuTcpSocket mysocket;
...
int16 length;
NU_Get_Peer_Name( mysocket.GetSocketDescriptor(),
                  (struct addr_struct*)socketAddress1,
                  &length );

// You wouldn't actually do this, instead you
// would call mysocket.GetPeerAddress which is
// much friendlier.
```



SocketAddress::Set

```
inline
VOID*
SocketAddress::Set
(
const IPAddress& ip,
  UINT16 uPort,
const CHAR* pName
= NULL
);
```

Sets IP address, port number and name.

Overview

Condition	Description
Pre-condition	None
Action	All internal data members are set.
Post-condition	None

Parameters

Parameter	Overview
ip	A reference to an IPAddress object.
uPort	Is an unsigned 16 bit integer that contains the new port number.
pName	A pointer to the name string corresponding to this <code>socketAddress</code> .

Return Value

None

Example

```
IPAddress ip(200,100,50,0);
SocketAddress address;

address.Set(ip,80);

address.Set(ip,80); // default name is NULL
```



SocketAddress::SetFamily

```
inline
VOID
SocketAddress::SetFamily
(
    INT16 nNewFamily
);
```

Called to set the network family member for a given `SocketAddress` object. The default value for this is `NU_FAMILY_IP`, which is the only currently supported protocol, so you don't need to worry about setting this

Overview

Condition	Description
Pre-condition	None
Action	Sets the family member to the passed in value.
Post-condition	None

Parameters

Parameter	Overview
nNewFamily	Contains a 16 bit integer representing the desired networking family. Note that currently the only valid value for this member is <code>NU_FAMILY_IP</code> , which is the default value. (that means you shouldn't have to worry about setting it).

Return Value

None

Example

```
SocketAddress sockaddr;
sockaddr.SetFamily( NU_FAMILY_IP );
```



SocketAddress::SetIP

```
inline  
VOID  
SocketAddress::SetIP  
(  
    UINT32 u32IP  
);
```

Called to set the IP address contained in the `SocketAddress` object by a 32 bit number.

Overview

Condition	Description
Pre-condition	The object exists.
Action	Does a 32 bit assignment of the value to the internal 32 bit IP address.
Post-condition	None

Parameters

Parameter	Overview
u32IP	Contains the 32 bit IP address.

Return Value

None

Example

```
SocketAddress address;  
address.SetIP( 12345678 );
```



SocketAddress::SetIP

```
inline
VOID
SocketAddress::SetIP
(
const IPAddress* pIP
);
```

Called to set the IP address from a pointer to an `IPAddress` object. No checking for NULL is done.

Overview

Condition	Description
Pre-condition	The object exists.
Action	Calls the <code>SetIP(UINT32)</code> overload.
Post-condition	None

Parameters

Parameter	Overview
<code>pIP</code>	A non-NULL pointer to an <code>IPAddress</code> object.

Return Value

None

Example

```
SocketAddress mysocket;
IPAddress    myip( 200, 100, 50, 0);
...
mysocket.SetIP( &myip );
```



SocketAddress::SetIP

```
inline  
VOID  
SocketAddress::SetIP  
(  
const IPAddress& ip  
);
```

Set the IP address by a reference to an `IPAddress` object.

Overview

Condition	Description
Pre-condition	The object exists.
Action	Calls the <code>SetIP(UINT32)</code> overload.
Post-condition	None

Parameters

Parameter	Overview
<code>ip</code>	A reference to an <code>IPAddress</code> object.

Return Value

None

Example

```
IPAddress ip(200,100,50,0);  
SocketAddress socketAddress;  
  
socketAddress.SetIP( ip );
```



SocketAddress::SetIP

```

inline
VOID
SocketAddress::SetIP
(
    UINT8 u1,
    UINT8 u2,
    UINT8 u3,
    UINT8 u4
);

```

Sets the IP address contained within the `SocketAddress` object by individual octets (bytes).

Overview

Condition	Description
Pre-condition	The object exists.
Action	Copies the given bytes individually into the internal data member.
Post-condition	None

Parameters

Parameter	Overview
u1	Contains the most significant byte of the IP address.
u2	Contains the second byte of the IP address.
u3	Contains the third byte of the IP address.
u4	Contains the least significant byte of the IP address.

Return Value

None

Example

```

SocketAddress *pSocketAddress = new SocketAddress;
pSocketAddress->SetIP( 200, 100, 50, 0 );

```



SocketAddress::SetName

```
inline  
VOID  
SocketAddress::SetName  
(  
    const CHAR* pName  
);
```

Called to set the pointer to the name for this desired `SocketAddress` object. Note that `SocketAddress` does not store the name, just a pointer to a name.

Overview

Condition	Description
Pre-condition	The object exists.
Action	Sets the internal name string pointer to the passed in name.
Post-condition	None

Parameters

Parameter	Overview
pName	A pointer to a string that contains the desired name. Note that this string must be a constant pointer, that is it must not go out of scope as only the pointer is saved, the name is not copied (This is done for compatibility with the C implementation underneath).

Return Value

None

Example

```
SocketAddress yoursocket;  
yoursocket.SetName( "YOURSERVER" );
```



SocketAddress::SetPort

```
inline
VOID
SocketAddress::SetPort
(
    UINT16 uPort
);
```

Called to set the port for a SocketAddress object.

Overview

Condition	Description
Pre-condition	None
Action	Sets the internal port member to the passed in value.
Post-condition	None

Parameters

Parameter	Overview
uPort	Contains an unsigned 16 bit integer for the desired port number.

Return Value

None

Example

```
SocketAddress mysocket;
mysocket.SetPort( 80 );
```



SocketAddress::SocketAddress

```
SocketAddress::SocketAddress  
(  
const SocketAddress& address  
);
```

Constructor. Normal copy constructor.

Overview

Condition	Description
Pre-condition	None
Action	Calls the assignment operator.
Post-condition	The object exists.

Parameters

Parameter	Overview
address	Reference to the SocketAddress object to copy.

Return Value

None

Example

```
SocketAddress servaddr;  
servaddr.SetIP(200,100,50,0);  
...  
SocketAddress servaddr2( servaddr );
```



SocketAddress::SocketAddress

```
SocketAddress::SocketAddress
(
const struct addr_struct& newSockaddr
);
```

Copy Constructor that takes a "C" addr_struct structure.

Overview

Condition	Description
Pre-condition	None
Action	Calls the assignment operator.
Post-condition	None

Parameters

Parameter	Overview
newSockaddr	SocketAddress object to copy.

Return Value

None

Example

```
struct addr_struct server_addr_struct;
...
SocketAddress myserver( server_addr_struct );
```



SocketAddress::SocketAddress

```
SocketAddress::SocketAddress  
(  
    const IPAddress& ip,  
    UINT16 uPort,  
    const CHAR* pName  
);
```

Constructor that takes an IP address, port and name.

Overview

Condition	Description
Pre-condition	None
Action	Calls the <code>Set (IPAddress, port, name)</code> member.
Post-condition	The object exists.

Parameters

Parameter	Overview
ip	Reference to an <code>IPAddress</code> object to copy.
uPort	Unsigned 16 bit Port number.
pName	A pointer to the name for the server. Note that only the pointer is stored, so this name must be persistent.

Return Value

None

Example

```
IPAddress ip{200,100, 50,1};  
...  
SocketAddress servaddr(ip, 7, "IMTRENDY.com");
```



SocketAddress : SocketAddress

```
SocketAddress::SocketAddress();
```

Default Constructor.

Overview

Condition	Description
Pre-condition	None
Action	Creates a SocketAddress object and initializes it with default values. IP address = 0.0.0.0 port = 0 name = NULL family = NU_FAMILY_IP
Post-condition	The object exists.

Parameters

None

Return Value

None

Example

```
SocketAddress servaddr;
```



